

# Bachelorarbeit

Benjamin Sass

Usability-Untersuchung von ausgewählten  
Testautomatisierungs-Werkzeugen für grafische  
Benutzeroberflächen

## Benjamin Sass

### Usability-Untersuchung von ausgewählten Testautomatisierungs-Werkzeugen für grafische Benutzeroberflächen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang Angewandte Informatik  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Bettina Buth  
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 16. Januar 2009

## **Benjamin Sass**

### **Thema der Bachelorarbeit**

Usability-Untersuchung von ausgewählten Testautomatisierungs-Werkzeugen für grafische Benutzeroberflächen

### **Stichworte**

Testautomatisierung, Usability

### **Kurzzusammenfassung**

Meine Arbeit beschäftigt sich mit Werkzeugen für den automatisierten Test von graphischen Benutzeroberflächen. Speziell sollen diese mittels Usability Tests auf ihre Benutzerfreundlichkeit getestet werden.

## **Benjamin Sass**

### **Title of the paper**

Usability analysis of selected test automation tools for graphical user interfaces

### **Keywords**

Testautomation, usability

### **Abstract**

This thesis is about tools for automated testing of graphical user interfaces. I want to analyse the usability of these tools in a usability lab.

## Inhaltsverzeichnis

Inhaltsverzeichnis.....	4
1 Einleitung.....	6
1.1 Inhaltlicher Aufbau der Arbeit.....	6
2 Automatisiertes Testen .....	7
2.1 Allgemein.....	7
2.2 Speziell bei grafischen Benutzeroberflächen .....	8
3 Marktübersicht.....	9
3.1 Auswahl der Werkzeuge.....	9
3.2 Kommerzielle Werkzeuge.....	10
3.2.1 Froglogic Squish for Java (Version 3.4.1).....	10
3.2.2 QFS QF-TEST (Version 2.2.4).....	11
3.3 Freie Werkzeuge.....	13
3.3.1 FEST (Version 1.0.1b1).....	13
3.3.2 Pounder (Version 0.95).....	15
3.3.3 jfcUnit (Version 2.08).....	16
4 Testfälle .....	17
4.1 Entwicklung / Programmierung eines einfachen Testlings .....	17
4.2 Erarbeitung der Aufgaben für die Usability Tests .....	18
4.3 Umsetzung mit den einzelnen Werkzeugen .....	19
4.3.1 Froglogic Squish.....	19
4.3.2 QFS QF-Test .....	23
4.3.3 FEST .....	25
5 Usability Tests.....	27
5.1 Definition von Usability .....	27
5.2 Ziel der Usability Tests.....	27
5.3 Erarbeitung eines Fragebogens für die Vorkenntnisse der Probanden .....	28
5.4 Durchführung der Tests im Usability Lab .....	29
5.4.1 Testumgebung.....	29
5.4.2 Arbeitsumgebung auf dem Testcomputer .....	29
5.4.3 Testpersonen.....	29
5.4.4 Testablauf.....	29
6 Auswertung der Usability Tests.....	31

6.1	Auswertungskriterien .....	31
6.2	Auswertung der Fragebögen .....	31
6.3	Auswertung für die einzelnen Produkte.....	32
6.3.1	QFS QF Test .....	32
6.3.2	Froglogic Squish.....	36
6.3.3	FEST .....	38
6.4	Mögliche Erklärung zum schlechten Abschneiden von QF-Test .....	41
6.5	Erkenntnisse aus den Tests .....	41
7	Vergleich aller Werkzeuge.....	42
7.1	Kommerziell.....	42
7.2	Frei.....	43
7.3	Fazit .....	44
8	Ausblick.....	45
9	Quellen .....	46
10	Anhang.....	47
10.1	Abbildungsverzeichnis.....	47
10.2	Fragebogen.....	48
10.3	Aufgabenzettel .....	49
10.4	Schnellstartanleitung für FEST.....	50
10.5	Testmetriken .....	53
10.5.1	Squish .....	53
10.5.2	QF-Test .....	53
10.5.3	FEST .....	53
10.6	Glossar.....	54

## 1 Einleitung

Grafische Benutzeroberflächen sind heute absoluter Standard bei Endanwendersoftware. Es gibt praktisch keine reinen Konsolenanwendungen mehr für den durchschnittlichen Computeranwender. Somit ist die grafische Oberfläche genauso wichtig wie die Algorithmen die im Hintergrund ablaufen.

In dieser Arbeit geht es mir darum solche Werkzeuge für den automatisierten Test grafischer Nutzeroberflächen vorzustellen und miteinander zu vergleichen.

Dazu möchte ich zum Mittel der Usability Tests greifen. Ich werde verschiedenen Werkzeuge auswählen und Probanden dann versuchen lassen festgelegte Aufgaben damit zu lösen. Ich will damit vor allem die Einsteigerfreundlichkeit der Werkzeuge bewerten, aber natürlich auch die Bedienung allgemein.

### 1.1 Inhaltlicher Aufbau der Arbeit

Im zweiten Kapitel werde ich mich kurz mit automatisierten Tests beschäftigen, erklären wie diese durchgeführt werden und was man damit erreichen kann.

Danach werde ich einige Tools am Markt vorstellen und kurz beschreiben.

In Kapitel vier geht es um die Erarbeitung der Aufgaben die die Probanden später im Usability Labor lösen sollen. Weiterhin zeige ich wie die Aufgaben mit ausgewählten Werkzeugen gelöst werden können.

Die eigentlichen Tests im Labor werden im fünften Kapitel behandelt.

Im sechsten Kapitel widme ich mich der Auswertung der Usability Tests. Ich zeige für alle drei ausgewählten Werkzeuge wo genau die Probleme lagen und wie viele Aufgaben die Probanden mit den jeweiligen Werkzeugen lösen konnten.

Zum Schluss folgt ein Vergleich der Werkzeuge auf abstrakter Ebene sowie ein kleiner Ausblick.

## 2 Automatisiertes Testen

### 2.1 Allgemein

Testautomatisierung ist die automatische Prüfung einer Software nach festgelegten Kriterien. Diese Kriterien müssen dabei in eine für den Computer verständliche Form gebracht werden.

Der Vorgang des automatisierten Testens lässt sich in folgende Tätigkeiten unterteilen:

#### **Testspezifikation**

In der Testspezifikation ist das Soll-Verhalten einer Anwendung beschrieben. Dies ist die Grundlage für die Tests.

#### **Testfallerstellung / Testscripterstellung**

Dieser Schritt umfasst das Anlegen der eigentlichen Tests. Dies ist der Schritt den die Probanden mit den gegebenen Werkzeugen durchführen sollen.

#### **Testdurchführung**

Die Durchführung wird vollautomatisch mit Werkzeugen wie z.B. JUnit durchgeführt. Hier ist es beispielsweise möglich, dass die Tests täglich nach dem nightly build ausgeführt werden.

#### **Testauswertung**

Bei der Auswertung wird das Ergebnis des vorigen Tests mit einem Sollwert verglichen. Weiterhin können auch zwei Software-Versionen miteinander verglichen werden um Verbesserungen und Verschlechterungen erkennen zu können.

#### **Testdokumentation**

Bei der Dokumentation wird die Testauswertung meist automatisiert zu einem Dokument zusammengefasst, dies kann z.B. eine HTML-Seite sein.

Der Begriff „automatisiertes Testen“ bezieht sich dabei auf die automatische Ausführung von Testscripten. Das generieren von Tests geschieht keineswegs automatisiert und ist immer die Aufgabe von Menschen.

Eine Form des automatisierten Testens ist beispielsweise JUnit unter JAVA. Hiermit ist es möglich z.B. Methoden aufzurufen und auf bestimmte Rückgabewerte zu kontrollieren.

Dies ist eine sehr effektive Art der um die Qualität der Software zu prüfen. Automatisierte Tests können beispielsweise bei jedem neuen Build angewendet werden um die Funktion aller bisher implementierten Features zu testen.

Es ist allerdings zu beachten, dass solche Tests nur die Anwesenheit von Fehlern beweisen können, aber niemals deren Abwesenheit. Diese Tests können bei einer nicht trivialen Software niemals alle Fehler finden, da es nicht möglich alle möglichen Kombinationen von Eingaben zu testen.

## 2.2 Speziell bei grafischen Benutzeroberflächen

Die Prüfung von GUIs ist eine spezielle Form von automatisierten Tests. Hier greift das Testwerkzeug direkt auf die GUI des Testlings zu. Man hat also die Möglichkeit automatisiert die grafischen Elemente zu bedienen. Diese Aktionen werden in ein Testscript eingetragen.

Checks ermöglichen die Abfrage von Werten in der GUI. Dies geschieht in dem man etwa den Text in einem Eingabefeld ausliest und mit einer Vorgabe vergleicht.

Man kann also komplette Benutzerinteraktionen mit der GUI hinterlegen (z.B. Anlegen eines Kunden) und prüfen ob der Vorgang der jeweiligen Version der Software problemlos ausgeführt werden konnte.

Komfortable Werkzeuge bieten bei solchen Tests die Möglichkeit die Interaktion mit der GUI des Testlings direkt aufzuzeichnen. Somit ist es dann nicht notwendig die gesamte GUI Interaktion per Hand in die Testsuite einzugeben.

Die Programmierung eines solchen Werkzeugs ist um einiges komplizierter als z.B. JUnit. Dies gilt vor allem für die Werkzeuge die nicht direkt in den Testling integriert werden. Hier wird der Testling meist im Kontext des Testwerkzeugs gestartet und alle Klicks des Benutzers auf die GUI des Testlings müssen abgefangen werden um sie aufzeichnen zu können. Desweiteren muss das Werkzeug in der Lage sein die GUI des Testlings zu bedienen, also Benutzerinteraktionen auszulösen. Diese Zugriffsmethoden sind je nach Art des Testlings sehr verschieden und man muss genau schauen ob das Testwerkzeug das eigene Toolkit unterstützt. In diesem Zusammenhang ist es nicht nur wichtig, dass das Werkzeug Java unterstützt sondern auch ob man Swing oder SWT für die GUI verwendet.



## 3 Marktübersicht

### 3.1 Auswahl der Werkzeuge

Für die Marktübersicht habe ich insgesamt fünf Produkte herausgesucht. Ich habe dabei einige Kriterien zugrunde gelegt um die Menge an Werkzeugen einzugrenzen.

Die Werkzeuge sollen in der Lage sein die GUI von Java-Swing Anwendungen zu prüfen. Ich habe in allen kommerziellen Fällen eine Demo oder Testversion des jeweiligen Herstellers verwendet. Dabei habe ich darauf geachtet, dass die Einschränkungen den Test nicht beeinflussen. Falls es mir also nicht möglich war eine Testversion mit ausreichendem Funktionsumfang zu erhalten, entspricht das Produkt ebenfalls nicht meinen Anforderungen. Außerdem ist auch für eine Firma, die dieses Produkt einsetzen will, vorteilhaft wenn sie vorher die Möglichkeit hat dieses ausgiebig zu testen, gerade in der Hinsicht auf die hohen Anschaffungskosten.

Ich habe zwei kommerzielle und drei open-Source Werkzeuge ausgewählt um einen Vergleich in Hinblick auf Funktionsumfang und Usability durchführen zu können. Dies ist vor allem in Hinsicht auf die hohen Preise der kommerziellen Produkte (mehrere Tausend Euro pro Lizenz) interessant.

## 3.2 Kommerzielle Werkzeuge

### 3.2.1 Froglogic Squish for Java (Version 3.4.1)<sup>1</sup>

Squish ist ein eigenständiges Programm. Ein Assistent unterstützt die Einbindung des Testlings.

Die Tests können hier in verschiedenen Scriptsprachen formuliert werden. Zur Auswahl stehen TCL, Python, JavaScript und Perl.

Die Unterstützung von zahlreichen Toolkits ermöglicht den Test von Programmen in verschiedensten Programmiersprachen. Unterstützt werden hier Qt, Tk, Web, Java (SWT und Swing) und Tq.

Die GUI Aktivität des Benutzers kann automatisch aufgezeichnet werden und wird dann in der gewählten Scriptsprache angezeigt.

Die Erzeugung von VerificationPoints (VP) erfolgt weitestgehend automatisch, es braucht nur die zu Prüfende Komponente und die zugehörige Eigenschaft gewählt zu werden.

Gerade am Start nimmt dieses Werkzeug den Benutzer „an die Hand“ und unterstützt ihn beim Einbinden des Testlings und der Aufzeichnung des ersten Tests.

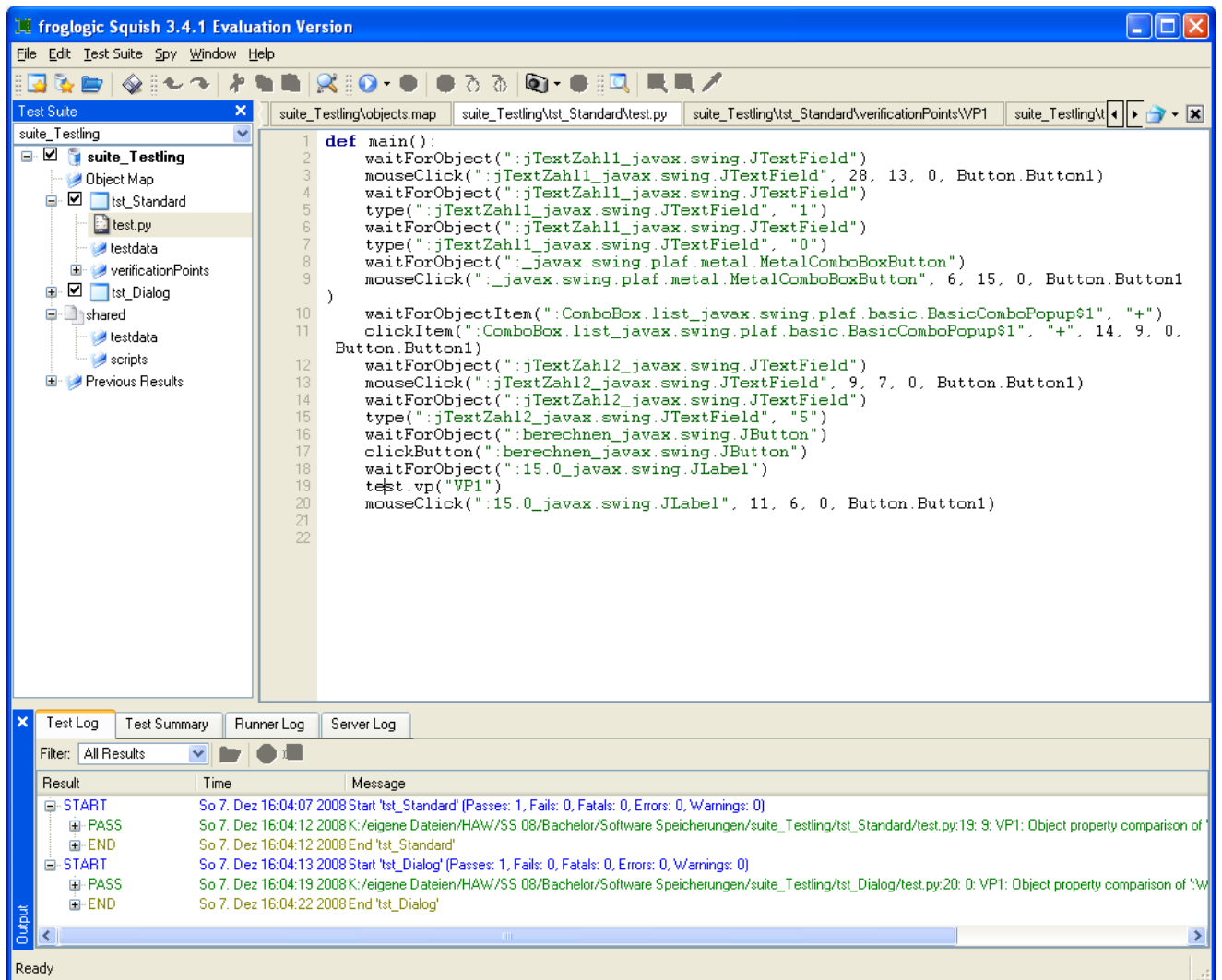


Abbildung 1: Froglogic Squish

<sup>1</sup> Download siehe Quelle (Froglogic)

### 3.2.2 QFS QF-TEST (Version 2.2.4)<sup>2</sup>

Bei QF-Test handelt es sich um eine eigenständige Software die installiert werden muss. Die notwendige Instrumentierung der JRE wird automatisch vorgenommen, muss allerdings bei einem Versionswechsel der JRE manuell eingeleitet werden. Die Instrumentierung ist notwendig um dem Werkzeug den Zugriff auf die GUI zu ermöglichen.

Es können eine Vielzahl von verschiedenen Anwendungstypen getestet werden, z.B. EXE-Dateien, JAVA Webstart Anwendungen, Applets in Browser oder Java Klassen. Für uns eignet es sich am Besten das kompilierte JAR-File einzubinden. Das Werkzeug unterstützt dabei die Einbindung des Testling mit einem praktischen Wizard.

Für die Testfallerstellung steht eine einfach zu bedienende Aufnahmefunktion bereit. Die einzelnen Schritte der Aufzeichnung werden übersichtlich in einer Baumstruktur dargestellt und lassen sich einzeln in einem Eigenschaftfenster bearbeiten.

Auch das Erstellen der Checks geschieht komfortabel indem man direkt die abzufragende Komponente in der GUI anklickt. Zusätzlich wird der aktuelle Wert der Komponente als Vergleichswert übernommen. Alle Angaben lassen sich aber natürlich auch wieder einfach im Testwerkzeug ändern.

Ein Debugger ermöglicht das komfortable Überprüfen der Testscripte. Auch das Setzen von Breakpoints ist möglich.

Als Auswertungsmöglichkeit bietet das Werkzeug übersichtliche HTML / XML Testberichte an.

---

<sup>2</sup> Download siehe Quelle (QFS)



## 3.3 Freie Werkzeuge

### 3.3.1 FEST (Version 1.0.1b1)<sup>3</sup>

FEST ist eine in Java geschriebene Testbibliothek die in das Projekt eingebunden werden muss (ähnlich JUnit). Dadurch bedingt werden alle Tests in Java programmiert, dies erhöht die Lesbarkeit für Java-Programmierer.

Über die mitgelieferte Klasse FrameFixture wird die GUI des Testlings aufgerufen und es wird ein Handle erzeugt über den die Befehle an die Komponenten der GUI gesendet werden können.

Für viele Standard GUI-Komponenten werden bereits passende Methoden mitgeliefert, die das Ansprechen der Komponenten sehr einfach gestalten. Die Referenzierung erfolgt direkt über deren Namen.

In diesem Beispiel wird in die Textbox „JeditZahl1“ der String „15“ eingetragen:

```
window.textBox("JeditZahl1").enterText("15");
```

Auch die Checks sind leicht zu verstehen. In diesem Fall wird der Wert der oben genannten TextBox abgefragt.

```
window.textBox("JeditZahl1").requireText("15");
```

Falls dieser nicht identisch mit dem tatsächlichen Wert ist wird der Test als nicht bestanden markiert.

Falls der Name einer Komponente nicht gefunden wird, wird automatisch eine einfache Komponentenhierarchie der GUI ausgegeben. Darin lässt sich mit ein wenig Mühe die gesuchte Komponente finden; dies wird allerdings problematischer je umfangreicher die GUI ist.

Da eine gute Kenntnis des Programms (z.B. den Namen der GUI-Komponenten) absolut von Vorteil ist, eignet es sich besonders für die Entwicklertests.

Zusätzlich zu den FEST Bibliotheken habe ich ein Netbeans Plug-In installiert, dass beim Anlegen eines Tests eine einfache Vorlage öffnet, die sich leicht für eigene Tests modifizieren lässt.

---

<sup>3</sup> Download siehe Quelle (FEST)

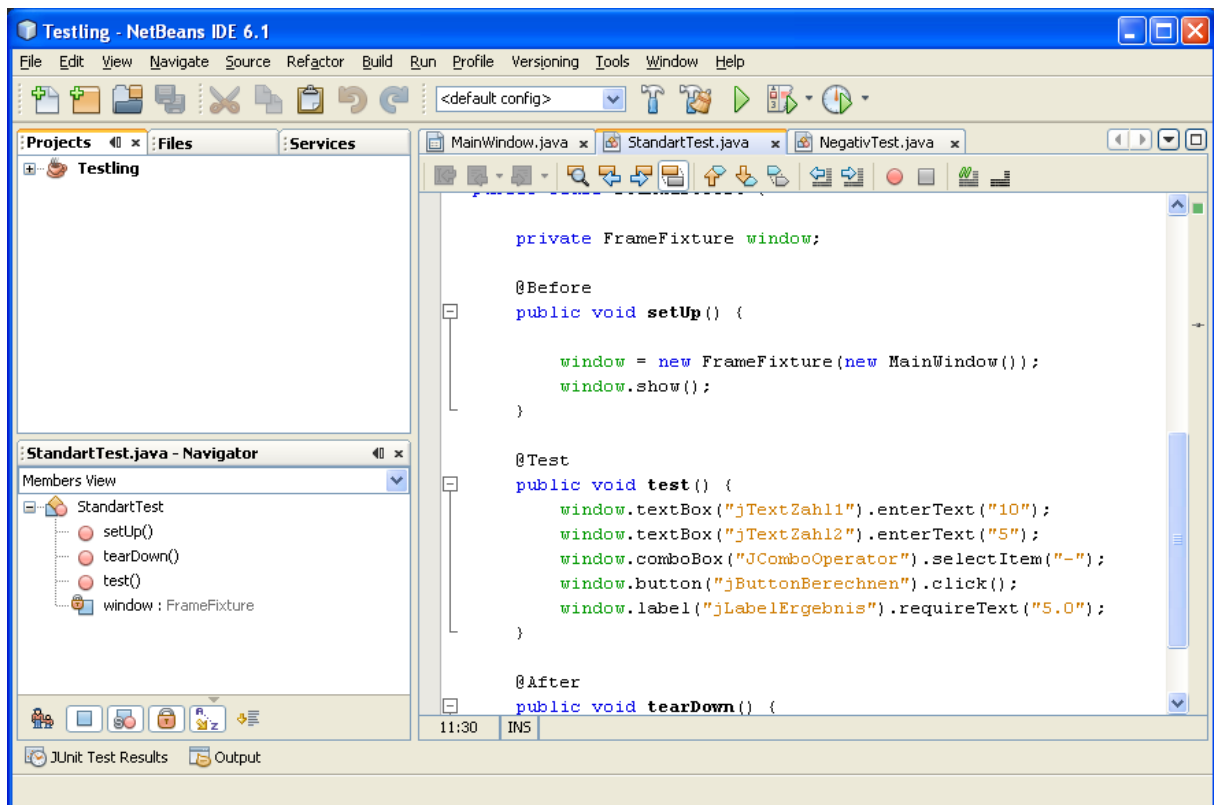


Abbildung 3: Ein FST-Test in Netbeans

### 3.3.2 Pounder (Version 0.95)<sup>4</sup>

Pounder ist ein sehr einfaches Werkzeug. Es besitzt im Gegensatz zu den anderen open-Source tools zwar eine eigene GUI aber der Funktionsumfang ist sehr gering.

Es ist gerade mal möglich eine Klasse anzugeben und dann die Mausbewegung sowie die Klicks auf der GUI aufzuzeichnen. Das aufgezeichnete Script lässt sich nur wiedergeben; das Abprüfen von Ergebnissen ist nicht möglich.

Die Weiterentwicklung von Pounder wurde bereits vor einiger Zeit eingestellt. Es dient in dieser Arbeit nur als Beispiel für ein anderes Konzept.

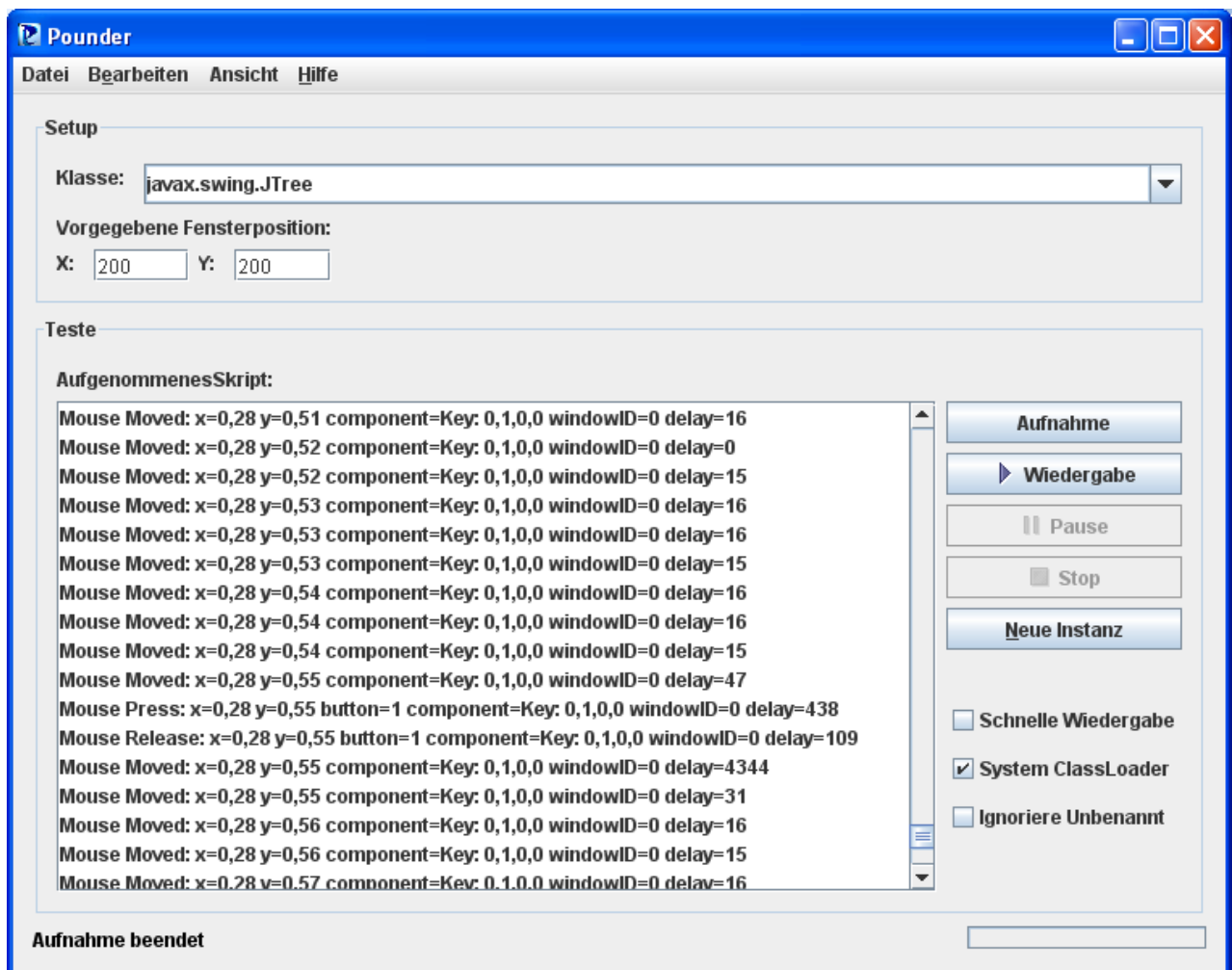


Abbildung 4: Pounder

Das aufgezeichnete Script ist sehr schlecht lesbar und sehr lang. Das liegt zum Einen daran, dass jede kleine Mausbewegung aufgezeichnet wird und zum Anderen, weil z.B. keine Namen von Komponenten im Script auftauchen.

<sup>4</sup> Download siehe Quelle (Pounder)

### 3.3.3 jfcUnit (Version 2.08)<sup>5</sup>

Dieses Werkzeug ist FEST sehr ähnlich. Es basiert ebenfalls auf JUnit und besitzt keine eigene GUI. Allerdings ist der Zugriff auf die Komponenten hier nicht so elegant gelöst:

```
NamedComponentFinder finder = new NamedComponentFinder(
    JComponent.class, "jTextZahl1");
JTextField zahl1 = ( JTextField ) finder.find( MainWindow, 0 );
assertNotNull( "Kann die Komponente jTextZahl1 nicht finden!", zahl1
);
getHelper().sendString( new StringEventData( this, zahl1, "15" ) );
```

Dieses Script sucht eine Komponente mit dem Namen „jTextZahl1“ und trägt dort den Wert „15“ ein. Um das gleiche in FEST zu erreichen genügt folgende Zeile:

```
window.textBox("JeditZahl1").enterText("15");
```

Da jfcUnit keine eigenen Assert-Methoden verwendet (wie z.B. die Require-Methoden in FEST) ist das Testscript auch in diesem Punkt unübersichtlicher.

Auch jfcUnit wird offenbar seit einigen Jahren nicht weiterentwickelt.

---

<sup>5</sup> Download siehe Quelle (jfcUnit)



## 4 Testfälle

### 4.1 Entwicklung / Programmierung eines einfachen Testlings

Der Testling ist das von den Probanden zu testende Programm. Er besteht aus einer einfachen Java Swing GUI, die einen sehr einfachen Taschenrechner realisiert. In die beiden Textfelder werden die Operanden eingegeben und in der Combobox ein Operator ( + , - , \* , / ) gewählt. Durch einen Druck auf den „Berechnen“ Button wird das Ergebnis der Berechnung in einem Label ausgegeben.

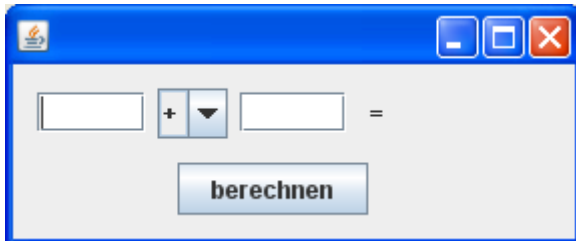


Abbildung 5: GUI des Testlings

Außerdem wird ein Hinweisdialog geöffnet wenn das Ergebnis der Berechnung negativ ist.

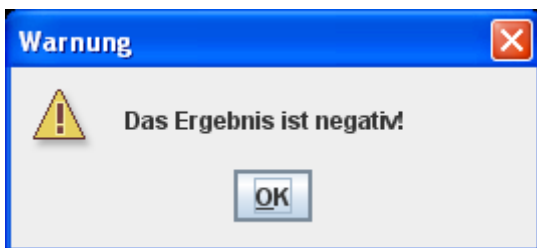


Abbildung 6: Hinweisdialog des Testlings bei negativen Ergebnis

Der Testling ist bewusst einfach gehalten, da sich der Proband gar nicht lange mit dem Testling beschäftigen soll, so dass er seine volle Aufmerksamkeit den Testwerkzeugen widmen kann. Allerdings enthält der Testling alle wichtigen Komponenten um auch Tests für größere Applikationen schreiben zu können. Dazu gehört die automatisierte Eingabe von Daten in Textboxen sowie die Bedienung einer Combobox. Als weitere Standardkomponente muss ein Button betätigt werden.

Für die Checks muss ein Label ausgelesen werden, der Hinweisdialog erkannt und weggeklickt werden.

## 4.2 Erarbeitung der Aufgaben für die Usability Tests

Um eine Vergleichbarkeit zwischen den Produkten zu ermöglichen, soll der Proband mit allen drei Werkzeugen die gleichen Aufgaben lösen.

Ziel der Aufgaben ist es grundlegende Funktionen der Werkzeuge kennenzulernen. Wenn eine Person alle Aufgaben erfolgreich bearbeiten konnte, hat sie das Werkzeug insoweit verstanden, dass sie in der Lage ist Tests für eigene Anwendungen zu schreiben.

Die erste Aufgabe überprüft ob der Proband in der Lage ist, den Testling in das Werkzeug einzubinden und Klicks auf diesen aufzuzeichnen. Zusätzlich soll auch eine Überprüfung stattfinden, da diese existenziell wichtig für alle automatisierten Tests ist. Weiterhin soll in der Aufgabe das ganze auf den Multiplikationsoperator erweitert werden. Dies zeigt ob die Testperson das Prozedere beim Additionsoperator vollständig verstanden hat und reproduzieren kann.

In der zweiten Aufgabe geht es hauptsächlich darum zusätzliche Komponenten in den Test einzubinden. Das ist in diesem Fall ein Dialogfeld.

Die dritte Aufgabe soll dem Probanden den Sinn der Tests vor Augen führen, indem er einen Fehler im Programm nachweist. Diese Aufgabe stellt dabei keine weitergehenden Ansprüche an den Probanden und sollte innerhalb kürzester Zeit erledigt werden können.

Der genaue Aufgabenzettel befindet sich im Anhang (Kap. 10.3: AufgabenzettelAufgabenzettel).

### 4.3 Umsetzung mit den einzelnen Werkzeugen

Im Folgenden möchte ich zeigen wie die Aufgaben mit den einzelnen Werkzeugen zu lösen sind. Ich habe dabei auf zwei Werkzeuge aus Kapitel 3 verzichtet. Pounder wird zum Einen nicht weiterentwickelt und zum Anderen unterstützt es keine Checks. Somit wären die Aufgaben hiermit nicht lösbar.

Auch verzichte auch den Einsatz von jfcUnit, da es FEST sehr ähnlich ist und weil die Testscripte wesentlich unübersichtlicher sind.

#### 4.3.1 Froglogic Squish

Beim Anlegen einer neuen Testsuite wird man zuerst gefragt welche Scriptsprache benutzt werden soll um die Testscripte darzustellen. Welche man hier wählt ist Geschmacksache. Danach folgt schon die Auswahl des Testlings, in diesem Fall unsere Testling.jar.

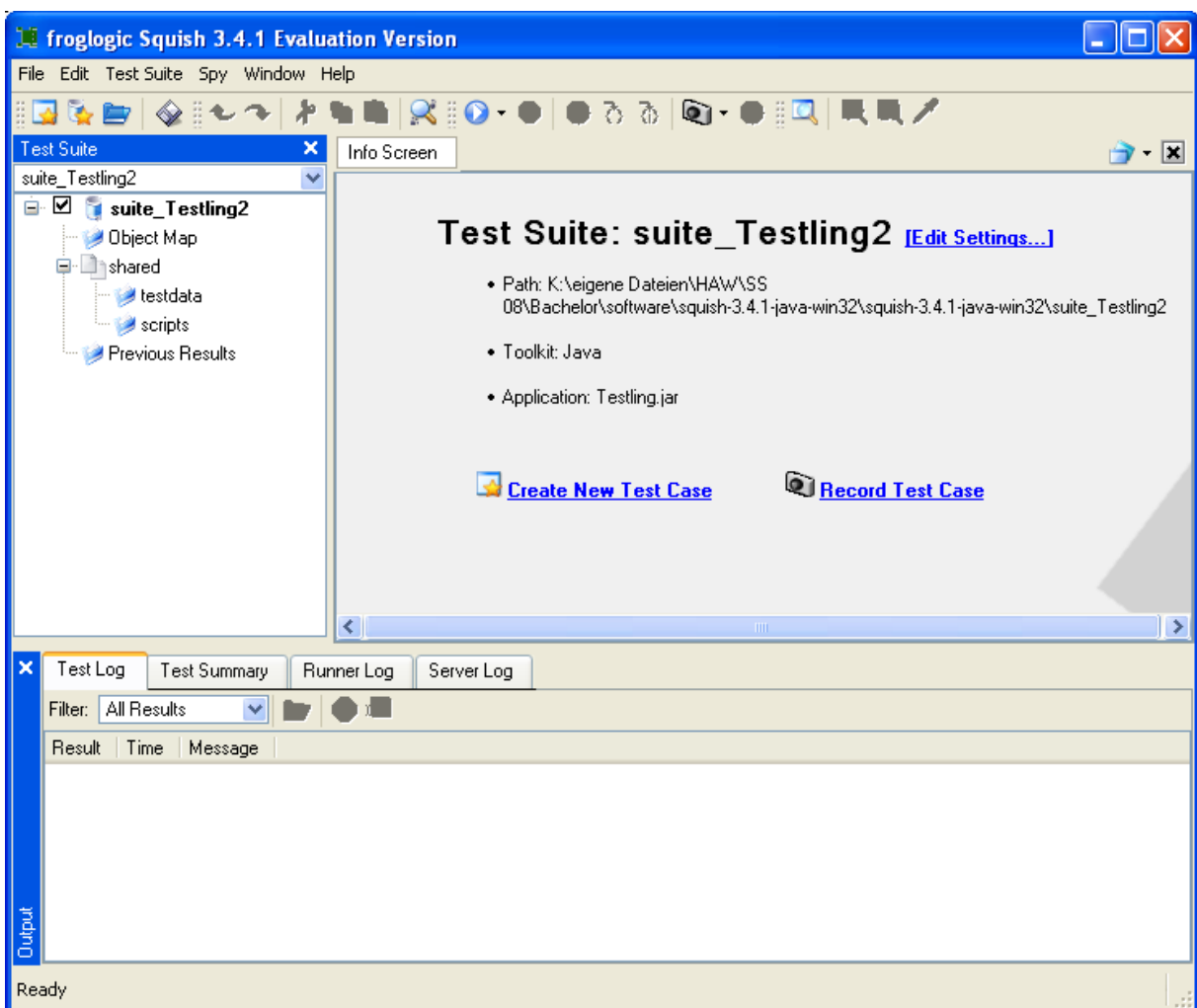


Abbildung 7: Begrüßungsbildschirm von Squish

Der nun folgende Begrüßungsbildschirm zeigt die wichtigsten Einstellungen noch einmal an und bietet den Einstieg in das Anlegen eines Testfalls. Hier wähle ich „Record Test Case“ um eine Benutzerinteraktion mit der GUI aufzuzeichnen. Nach Angabe eines Namens für den Testfall startet der Testling und man tippt zwei Operanden ein und klickt auf „berechnen“. In der daneben angezeigten „Squish Control Bar“ kann man erkennen was aufgezeichnet wird.

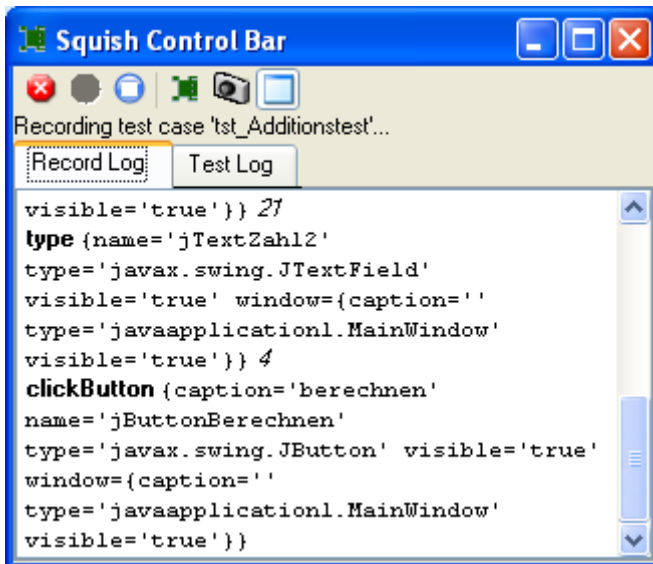


Abbildung 8: Squish Control Bar

Um die Aufnahme zu Beenden klickt man nun auf das Stoppzeichen. Daraufhin wird der Testfall in der Testsuite angelegt. Nun folgt das Anlegen des Checks, dazu wird in die letzte Zeile des Testscripts ein Breakpoint gesetzt und das Script gestartet. Das führt in diesem Fall allerdings zu dem Problem, dass dann der Druck auf den Berechnen Knopf noch nicht ausgeführt wurde und somit das Label noch gar nicht sichtbar ist. Also greift man zu einem kleinen Trick und kopiert die letzte Zeile.

Ein anderer Weg wäre hier einfach das Label im „Spy“ per Hand auszuwählen.

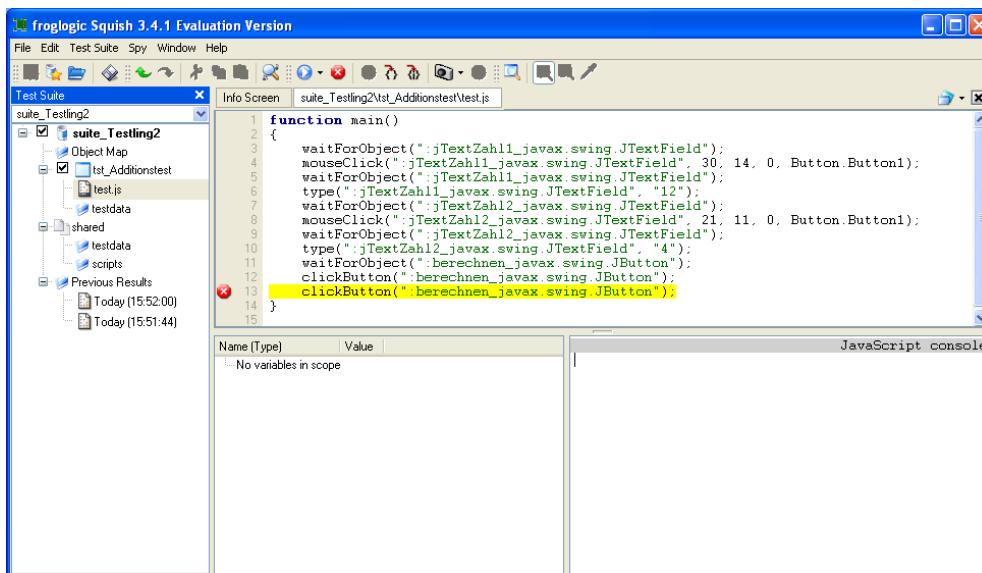


Abbildung 9: Breakpoint im Testscript zum Anlegen des VP

Nun wird der „Spy“ gestartet und durch einen Klick auf das Pipettensymbol wird der „Object Picker“ gestartet mit dessen Hilfe wir nun das Label mit dem Ergebnis auf der GUI des Testlings anklicken können.

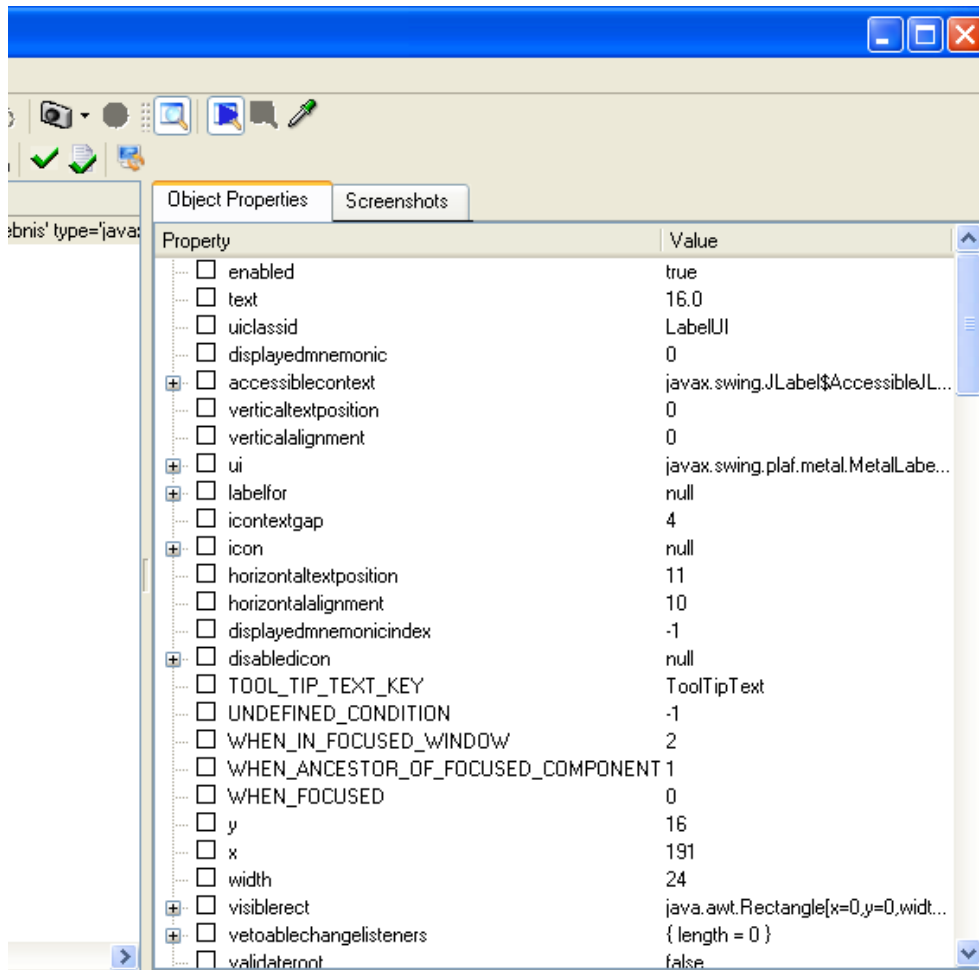


Abbildung 10: Attribute des Labels im Spy

Der Spy zeigt nun viele Eigenschaften der Label-Komponente an. Da uns in diesem Fall der Inhalt des Labels interessiert wählt man die Checkbox vor „text“ aus. Durch einen Klick auf den grünen Haken wird der VP angelegt und ein Verweis dazu im Testscript hinterlegt.

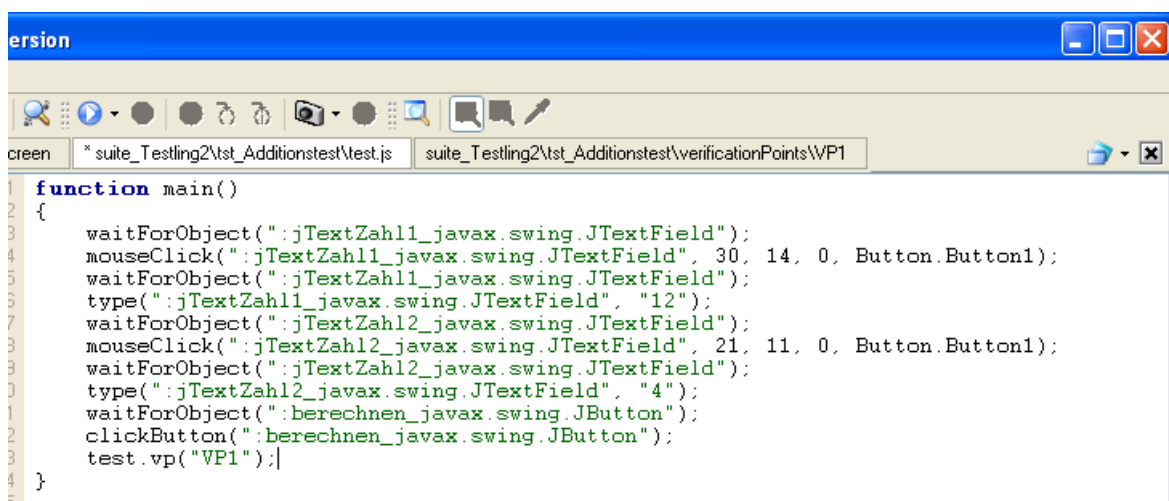
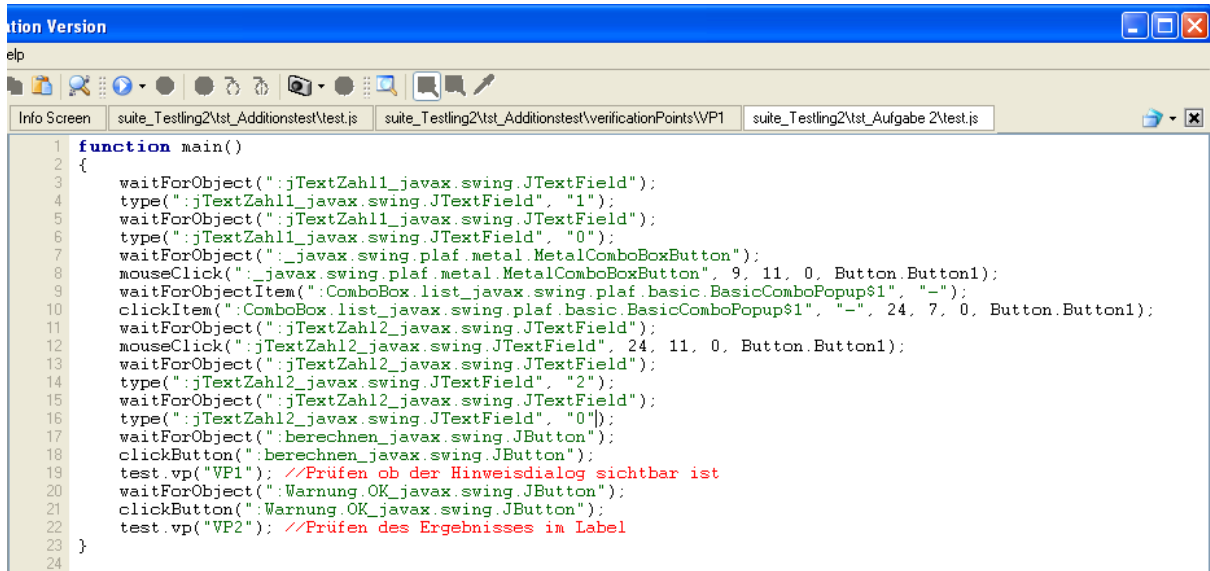


Abbildung 11: Kompletter Additionstest inklusive VP

Analog dazu wird nun verfahren um den Multiplikationstest für Aufgabe 1 anzulegen. Für die zweite Aufgabe zeichnet man wieder einen einfachen Test auf, diesmal mit einem negativen Ergebnis.

Zusätzlich muss hier noch der Hinweisdialog weggeklickt werden. Um die Aufgabe zu erfüllen werden zwei VPs benötigt. Der erste(VP1) prüft ob der Hinweisdialog erscheint, dies kann z.B. über die „visible“ Eigenschaft geschehen. Der zweite(VP2) prüft wieder das Ergebnis der Berechnung im Label.



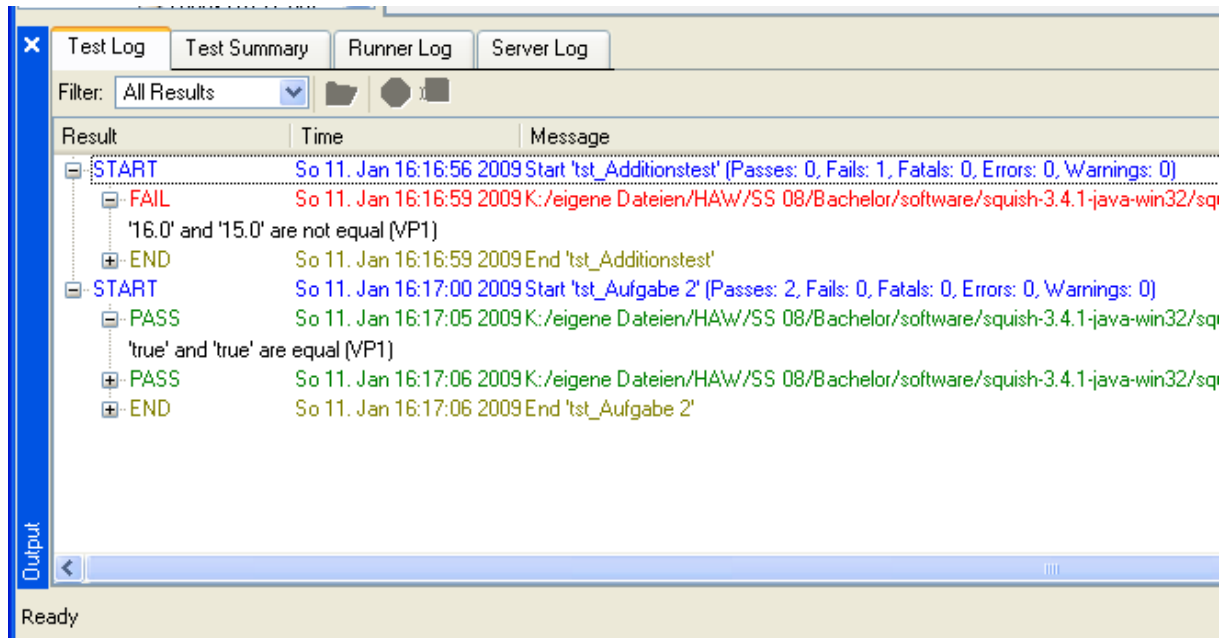
```

1 function main()
2 {
3     waitForObject(":jTextZahl1_javax.swing.JTextField");
4     type(":jTextZahl1_javax.swing.JTextField", "1");
5     waitForObject(":jTextZahl1_javax.swing.JTextField");
6     type(":jTextZahl1_javax.swing.JTextField", "0");
7     waitForObject(":_javax.swing.plaf.metal.MetalComboBoxButton");
8     mouseClicked(":_javax.swing.plaf.metal.MetalComboBoxButton", 9, 11, 0, Button.Button1);
9     waitForObjectItem(":ComboBox.list_javax.swing.plaf.basic.BasicComboPopup$1", "-");
10    clickItem(":ComboBox.list_javax.swing.plaf.basic.BasicComboPopup$1", "-", 24, 7, 0, Button.Button1);
11    waitForObject(":jTextZahl2_javax.swing.JTextField");
12    mouseClicked(":jTextZahl2_javax.swing.JTextField", 24, 11, 0, Button.Button1);
13    waitForObject(":jTextZahl2_javax.swing.JTextField");
14    type(":jTextZahl2_javax.swing.JTextField", "2");
15    waitForObject(":jTextZahl2_javax.swing.JTextField");
16    type(":jTextZahl2_javax.swing.JTextField", "0");
17    waitForObject(":berechnen_javax.swing.JButton");
18    clickButton(":berechnen_javax.swing.JButton");
19    test_vp("VP1"); //Prüfen ob der Hinweisdialog sichtbar ist
20    waitForObject(":Warnung_OK_javax.swing.JButton");
21    clickButton(":Warnung_OK_javax.swing.JButton");
22    test_vp("VP2"); //Prüfen des Ergebnisses in Label
23 }
24

```

Abbildung 12: Lösung der zweiten Aufgabe mit Squish

Für Aufgabe 3 sind keine weiteren Tests nötig, darum zeige ich hier nur den „Test Log“ mit den Ergebnissen aller Tests.



Result	Time	Message
START	So 11. Jan 16:16:56 2009	Start 'tst_Additionstest' (Passes: 0, Fails: 1, Fatafs: 0, Errors: 0, Warnings: 0)
FAIL	So 11. Jan 16:16:59 2009	K:/eigene Dateien/HAW/SS 08/Bachelor/software/squish-3.4.1-java-win32/sq '16.0' and '15.0' are not equal (VP1)
END	So 11. Jan 16:16:59 2009	End 'tst_Additionstest'
START	So 11. Jan 16:17:00 2009	Start 'tst_Aufgabe 2' (Passes: 2, Fails: 0, Fatafs: 0, Errors: 0, Warnings: 0)
PASS	So 11. Jan 16:17:05 2009	K:/eigene Dateien/HAW/SS 08/Bachelor/software/squish-3.4.1-java-win32/sq 'true' and 'true' are equal (VP1)
PASS	So 11. Jan 16:17:06 2009	K:/eigene Dateien/HAW/SS 08/Bachelor/software/squish-3.4.1-java-win32/sq
END	So 11. Jan 16:17:06 2009	End 'tst_Aufgabe 2'

Abbildung 13: Test Log nach der dritten Aufgabe

### 4.3.2 QFS QF-Test

Zuerst wird der Testling integriert, dies geschieht unter Extras->Schnellstart Wizard. Hier wählt man das JAR-Archiv aus.

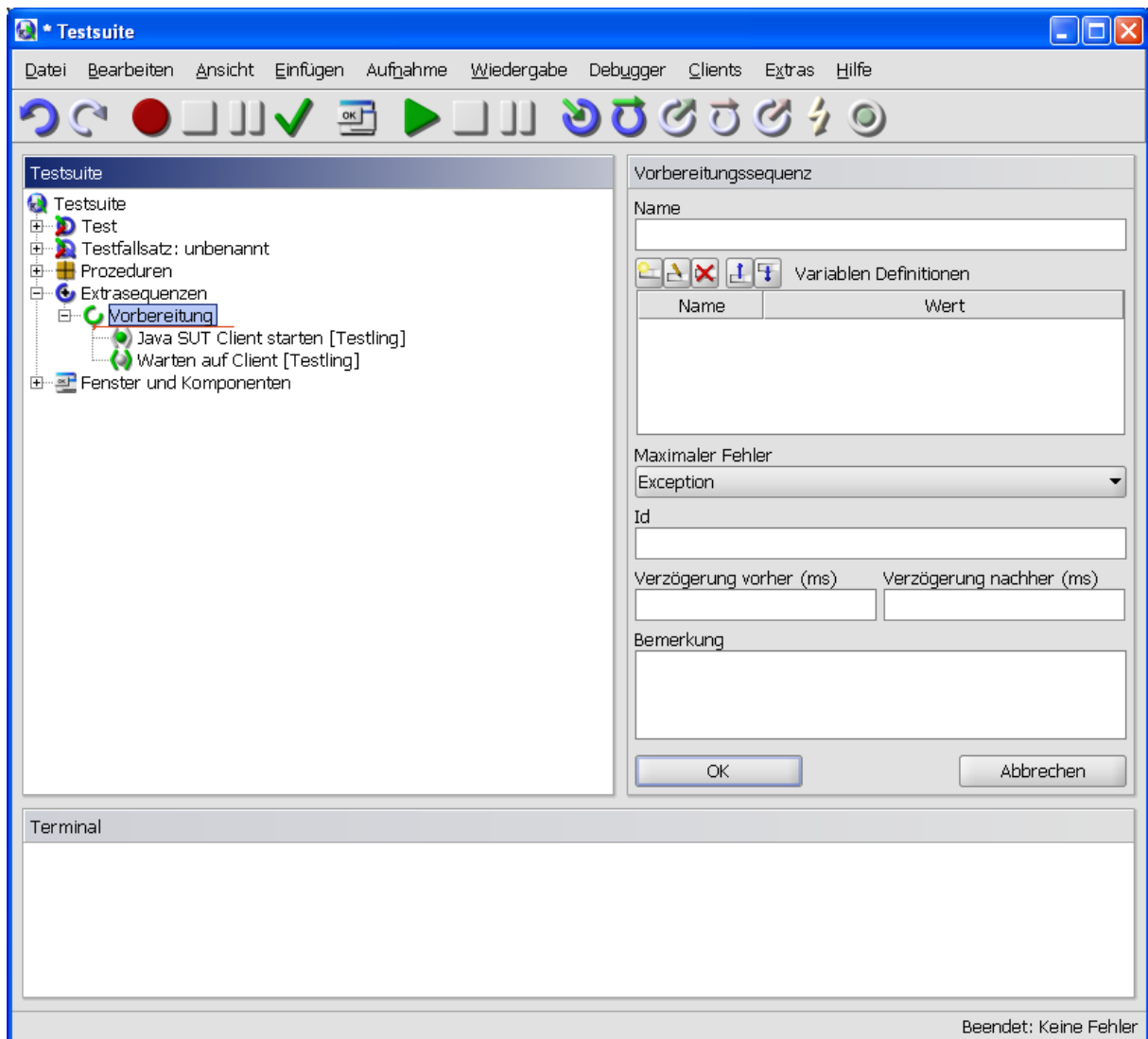


Abbildung 14: Neue Testsuite mit integriertem Testling

Nun kann man über den Aufnahmeknopf (roter Kreis) die Benutzerinteraktion für den ersten Test aufzeichnen. Danach klickt man auf den grünen Haken um einen Check anzulegen. Jetzt klickt man auf das Label, dass das Ergebnis enthält. Nun wird durch einen Druck auf den Stopp Knopf die Aufzeichnung beendet und eine Sequenz in der Testsuite angelegt.

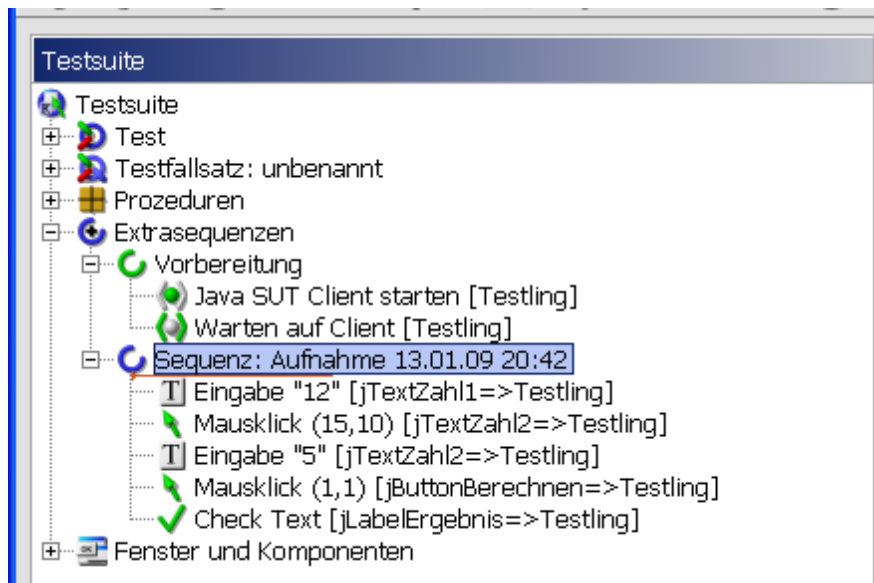


Abbildung 15: Testsequenz des Additionstests

Jetzt werden die aufgenommene Sequenz, der Start des Testlings und das Beenden des Testlings zu einem Test zusammengefasst.

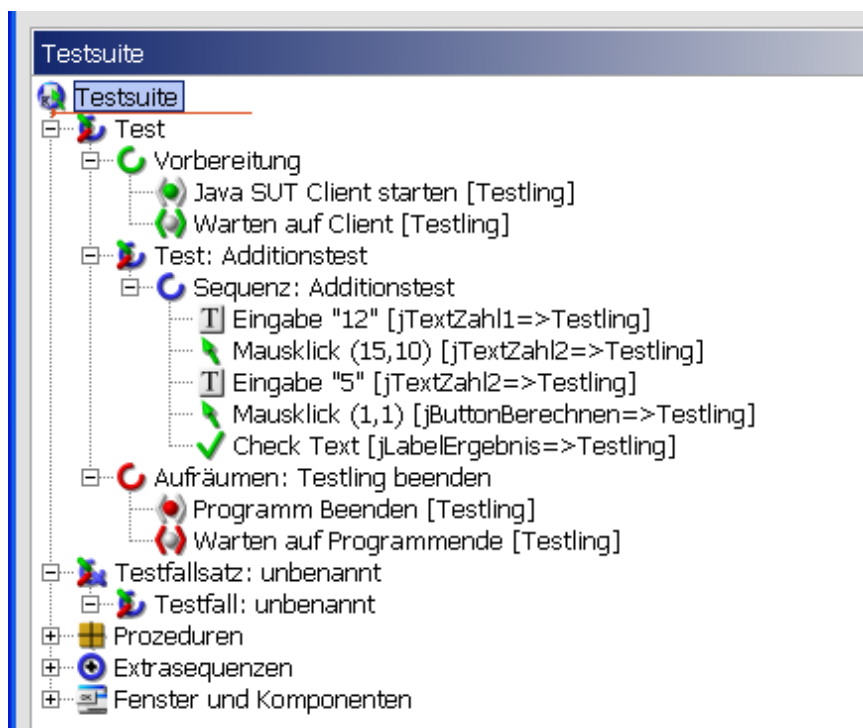


Abbildung 16: Additionstest in QF-Test

Da die folgenden Tests analog dazu ablaufen werde ich sie an dieser Stelle nicht weiter dokumentieren.



### 4.3.3 FEST

So sieht die Lösung für den Additionstest in Aufgabe 1 aus:

```
public class StandartTest
{
    private FrameFixture window;

    @Before
    public void setUp() {
        window = new FrameFixture(new MainWindow());
        window.show();
    }

    @Test
    public void Additionstest() {
        window.textBox("jTextZahl1").enterText("10");
        window.textBox("jTextZahl2").enterText("5");
        window.comboBox("JComboOperator").selectItem("+");
        window.button("jButtonBerechnen").click();
        window.label("jLabelErgebnis").requireText("15.0");
    }

    @After
    public void tearDown() {
        window.cleanup();
    }
}
```

Der Multiplikationstest ist sehr ähnlich. Allerdings ist dabei zu beachten, dass der Operator in der Combobox nicht über einen String gewählt werden darf, da dies zu einem Fehler führt. Stattdessen muss dieser über den Index gewählt werden. Dies ist zwar ein schlechter Stil, da bei jeder Veränderung der Reihenfolge der Operatoren in der Combobox auch die Tests angepasst werden müssen, aber leider gibt es in diesem Fall keine andere Möglichkeit.

```
window.comboBox("JComboOperator").selectItem(2);
```

Interessant ist allerdings noch die Lösung für die zweite Aufgabe. Da die restlichen Teile des Tests unverändert bleiben, stelle ich hier nur die eigentliche Testmethode dar. Wichtig ist hier die Überprüfung auf Vorhandensein des Hinweisfensters (`window.dialog.requireVisible`) sowie das darauf folgende wegklicken der Meldung.

```
@Test
public void aufgabe2() {
    window.textBox("jTextZahl1").enterText("5");
    window.textBox("jTextZahl2").enterText("10");
    window.comboBox("JComboOperator").selectItem("-");
    window.button("jButtonBerechnen").click();

    window.dialog().requireVisible();
}
```

```
    window.dialog().click();  
  
    window.label("jLabelErgebnis").requireText("-5.0");  
}
```

## 5 Usability Tests

### 5.1 Definition von Usability

Usability lässt sich im Deutschen mit verschiedenen Begriffen übersetzen. Die passendste Übersetzung ist dabei wohl „Gebrauchstauglichkeit“. Er impliziert, dass eine Software ohne optimierte Usability nicht brauchbar ist. Die Benutzer kämpfen also mit dem Aufbau und der Bedienung der Software und können von dessen eigentlichen Nutzen kaum profitieren.

Als „Software Ergonomie“ ist der Begriff auch in ISO 9241 definiert:

„Usability ist das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen.“

### 5.2 Ziel der Usability Tests

In meinen Tests geht es primär darum die Einsteigerfreundlichkeit der Produkte zu testen. Dies hat folgenden Grund: wenn eine Software nur wenig Einarbeitung fordert, dann wird diese auch eher im Betrieb eingesetzt, als wenn dem ein langer Lernprozess vorausgehen muss. Hierbei sind vor allem die Kosten von hoher Bedeutung, die durch die Schulungen entstehen.

Man stelle sich die Situation vor, dass ein Mitarbeiter von seinem Chef die Aufgabe bekommt ein Testwerkzeug für die Firma zu besorgen. Dieser wird sich dann wohl verschiedene Produkte aus dem Internet herunterladen und diese kurz austesten. Man kann dabei davon ausgehen, dass er das Produkt wählt mit dem er/sie in der kurzen Testzeit das Meiste erreicht hat. Somit halte ich die Einstiegsfreundlichkeit für ein sehr wichtiges Kriterium.

Um die Einsteigerfreundlichkeit zu ermitteln, werde ich mehreren Testpersonen die gleichen Produkte vorlegen. Dabei soll sicher gestellt werden, dass noch keiner von ihnen jemals mit diesen Produkten gearbeitet hat. Dies wird im Vorhinein durch den Fragebogen abgeprüft und dokumentiert.

### 5.3 Erarbeitung eines Fragebogens für die Vorkenntnisse der Probanden

Um die Einsteigerfreundlichkeit der Produkte bewerten zu können, ist es mir wichtig, dass die Probanden vorher noch nie ein Werkzeug für automatisierte GUI Tests verwendet haben. Dieses sichere ich durch eine Frage ab.

Weiterhin möchte ich von den Probanden wissen ob sie jemals automatisierte Tests (z.B. JUnit) geschrieben haben. Dies ist interessant, weil der dritte Test ein auf JUnit basierendes Werkzeug darstellt für den JUnit Erfahrungen sehr hilfreich sein können.

Ebenso ist es mir wichtig zu wissen wie die Testperson normalerweise an neue Programme herangeht. Ich möchte zum Einen erfahren, ob sie das Verhalten bei den Testpersonen ähnelt und zum Anderen ob sie sich während der Tests wie angegeben verhalten.

Der vollständige Fragebogen befindet sich im Anhang (Kap. 10.2).

## 5.4 Durchführung der Tests im Usability Lab

### 5.4.1 Testumgebung

Die Tests wurden im Usability Labor der HAW Hamburg durchgeführt. Das Labor besitzt verschiedene Geräte um das Verhalten des Probanden aufzuzeichnen:

- Sechs Kameras filmen die Testpersonen aus verschiedenen Blickwinkeln
- Zwei Mikrofone zeichnen die Kommentare des Probanden auf
- Ein Eyetracker zeichnet die Augenbewegung auf dem Bildschirm auf und blendet diese in die Aufnahme des Bildschirms ein
- Aufzeichnung aller Maus- und Tastatureingaben durch eine spezielle Software

Als Ergebnis der Tests erhalte ich ein Video, das die Bilder der sechs Kameras sowie das Bild des Eyetrackers enthält. Zusätzlich erhalte ich Diagramme, die die Anzahl Mausklicks, Tastaturanschläge etc. enthalten.

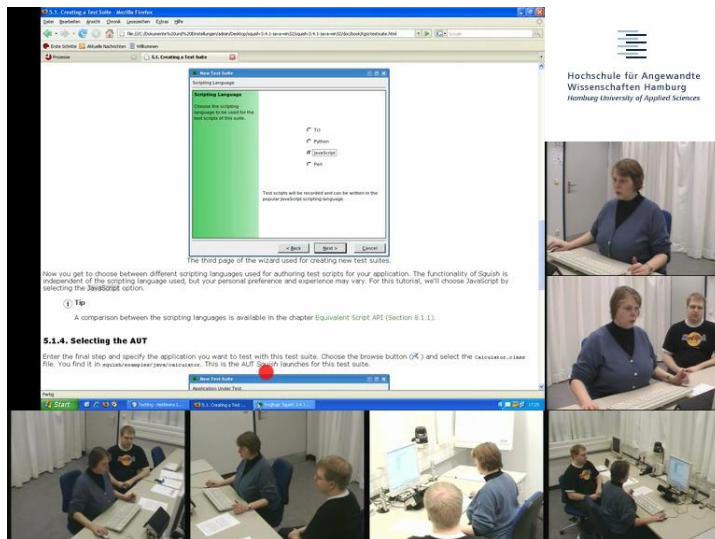


Abbildung 17: Video aus dem Usability Labor

### 5.4.2 Arbeitsumgebung auf dem Testcomputer

Bei allen Probanden war bereits die Netbeans IDE mit dem Testling Projekt und das zu testende Werkzeug geöffnet.

Für den dritten Test (FEST) ist ebenfalls das Netbeans Plug-In installiert, um den Probanden den Einstieg zu erleichtern. Zusätzlich erhielten die Probanden eine Schnellstartanleitung für FEST von der offiziellen Homepage(s. Kap. 10.4).

### 5.4.3 Testpersonen

Bei den Testpersonen handelte es sich um zwei Studenten des Bachelor Studienganges „Angewandte Informatik“ der HAW Hamburg sowie eine Professorin der HAW. Keine der Testpersonen hatte vorher an einem Test im Usability Labor teilgenommen.

### 5.4.4 Testablauf

Die Testpersonen hatten die Möglichkeit den Aufgabebettel vor Beginn des Tests vollständig zu lesen und ggf. Fragen zu klären. Weiterhin konnte sich jeder Proband mit der Funktionsweise des Testlings vertraut machen und auch hier ggf. Fragen stellen.

Während des Tests befand ich mich im gleichen Raum wie der Proband, so dass ich die Möglichkeit hatte in den Test einzugreifen (Teilnehmende Beobachtung). Ich habe dies so selten wie möglich getan, nur wenn ich das Gefühl hatte dass der Proband bei einem Problem über längere Zeit nicht weiterzukommen schien.

Die Tests wurden in folgender Reihenfolge durchgeführt:

- 1) QFS QF-Test
- 2) Froglogic Squish for JAVA
- 3) FEST

Pro Werkzeug hat jeder Proband genau 30 Minuten Zeit alle Aufgaben zu lösen.

Testperson 1 hat alle drei Tests hintereinander absolviert. Es wurden nur kurze Pausen von 5-10 Minuten zwischen den Tests eingelegt.

Die Testpersonen 2 und 3 absolvierten die Tests im Wechsel, so dass bei jedem auf 30 Minuten Test auch 30 Minuten Pause folgte.

Wenn der Proband mit einfachem Ausprobieren nicht zum Ziel kommt, wurde dieser angewiesen die im Programm enthaltene Hilfe zu verwenden. Direkte Hilfe über das Internet wurde den Probanden nicht gestattet.

## 6 Auswertung der Usability Tests

### 6.1 Auswertungskriterien

Für die Bewertung möchte ich folgende Kriterien zugrunde legen:

- Wie viele Aufgaben konnte der Proband lösen
- Braucht der Proband die Hilfefunktion oder findet er/sie sich selbst zurecht
- Musste ich bei dem Test eingreifen
- Nach welcher Zeit konnte die erste Aufgabe gelöst werden
- Auf welche Probleme stößt der Proband während des Tests

Das wichtigste Kriterium ist selbstverständlich die Anzahl der gelösten Aufgaben. Es liegt auf der Hand, dass der Proband umso besser mit dem Werkzeug umgehen kann, je mehr Aufgaben er gelöst hat.

Weiterhin ist es wichtig, ob es das Werkzeug schafft, sich selbst zu erklären. Je selbsterklärender es ist, desto weniger muss der Proband in die Hilfe schauen.

Wenn ich in einen Test eingreifen muss, dann ist dies prinzipiell erstmal negativ, aber es war teilweise notwendig, damit die Probanden nicht schon am Anfang des Tests „steckenbleiben“. Wenn ich einem Probanden größere Hilfestellung leisten musste, dann ist dies unter den Problemen aufgeführt.

Die Einsteigerfreundlichkeit werde ich an der Zeit messen, die die Testperson gebraucht hat um die erste Teilaufgabe zu bewältigen. Dies ist also die Zeit, die der Benutzer braucht, um sich grundsätzlich im Programm zurecht zu finden.

Stößt der Proband während des Tests auf Probleme (z.B. bei der Bedienung) halte ich diese ebenfalls fest. Eines der wichtigsten Ziele solch einer Untersuchung sollte es sein, derartige Stolpersteine aus dem Weg zu räumen.

Im Anhang (s. Kap 10.5) befinden sich die Testmetriken. Dies sind z.B. Mausclicks, Tastaturanschläge etc. Meiner Meinung nach scheinen sie sich in diesem Fall nicht als Auswertungskriterium zu eignen, da sich keine Muster erkennen lassen.

### 6.2 Auswertung der Fragebögen

Alle drei Testpersonen antworten bei den Fragen eins und zwei gleich, somit hatten alle Erfahrungen mit JUnit, aber noch niemals Testautomatisierungswerkzeuge für GUIs verwendet.

Auch die Ergebnisse von Frage drei waren sehr ähnlich. Die Testpersonen versuchen zuerst sich selbst zurechtzufinden, führt dies nicht zum gewünschten Ergebnis, verwenden sie die integrierte Hilfe. Falls keine der Methoden helfen konnte suchen die Probanden Hilfe im Internet.

## 6.3 Auswertung für die einzelnen Produkte

Ich habe für jedes Werkzeug eine Tabelle erstellt, in der zu erkennen ist, welche Aufgaben die Probanden lösen konnten. Dabei habe ich folgende drei Symbole verwendet:

- Haken (✓): Die Aufgabe wurde vollständig gelöst
- Kreis (○): Die Aufgabe wurde bearbeitet aber nicht vollständig gelöst
- Kreuz (✗): Die Aufgabe wurde gar nicht erst begonnen

Ich habe in der Tabelle Aufgabe 1 in zwei Bereiche aufgeteilt (Additionstest & Multiplikationstest), um auch kleinere Fortschritte dokumentieren zu können. In der letzten Spalte habe ich die Zeit angegeben, die der Proband benötigte um die erste Teilaufgabe (Additionstest) zu lösen.

### 6.3.1 QFS QF Test

#### 6.3.1.1 Gelöste Aufgaben

Leider hat es kein Proband geschafft auch nur die erste Aufgabe vollständig zu lösen. Die Meisten Probleme hatten die Probanden damit, den Testling einzubinden. An diesem Punkt musste ich bei allen Probanden eingreifen, damit diese noch die Chance hatten ein Testscript aufzuzeichnen und Checks anzulegen.

Sie schafften es dann meist eine Interaktion mit der GUI aufzuzeichnen, aber keiner hat es geschafft, einen Check für das Ergebnis anzulegen.

Proband	Additionstest	Multiplikationstest	Aufgabe 2	Aufgabe 3	Zeit für Add-Test
1	○	✗	✗	✗	-
2	○	✗	✗	✗	-
3	○	✗	✗	✗	-

#### 6.3.1.2 Erkannte Probleme

1. Einbindung des Testlings in die Testsuite
  - Den Testpersonen ist nicht klar, wie sie den Testling in das Testwerkzeug integrieren können, um die Tests auszuführen.
2. Verwenden der Import Funktion um Testling einzubinden
3. Sinn der Include Dateien?
  - Die Probanden versuchten an dieser Stelle teilweise den Quellcode des Testlings einzubinden. Normalerweise können hier Tests aus anderen Testsuiten eingebunden werden.
4. Variablen bei Testfall
  - Testperson wollte dort durch Eingabe der Variablennamen aus dem Java Quellcode direkt auf diese zugreifen, hiermit sind allerdings Variablen innerhalb des Testscripts gemeint.
5. Probanden verstehen nicht, dass die Komponenten direkt angeklickt werden können
  - Die Probanden haben versucht die Komponenten des Testlings direkt anzugeben (z.B. als Variable) und haben erst spät den Weg gefunden die Interaktion direkt aufzeichnen zu lassen.



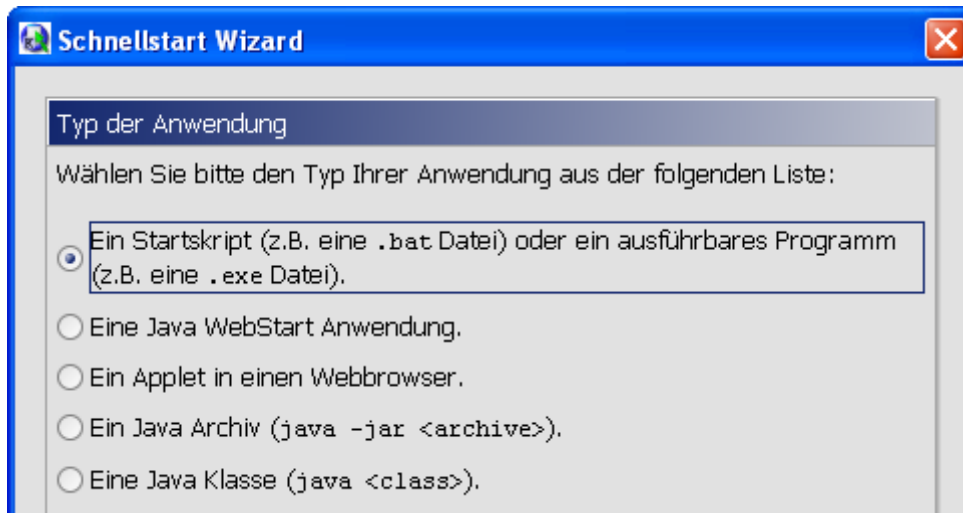


Abbildung 18: Auswahl des Typs der Anwendung in QF-Test

6. Verschiedene Arten den Testling zu starten (Programm, Java SUT, etc.)
  - Die Vielzahl an Möglichkeiten den Client zu starten, führte bei den Testpersonen dazu, dass sie nicht alle Einträge durchlesen und so einen unpassenden Eintrag wählten.
7. Datei öffnen Dialog sehr träge
  - Alle Dialoge zum Auswählen von Dateien brauchen recht lange bis sie geladen sind und auch das „durchhangeln“ durch den Verzeichnisbaum ist recht träge.



Abbildung 19: Undeutliche Fehlermeldung in QF-Test

8. Ungenaue Fehlermeldung
  - Es war den Testpersonen nicht ersichtlich welcher Wert nicht leer sein darf. Die Markierung des betreffenden Feldes wird nur durch die Positionierung des Cursors markiert.
9. Programm starten Dialog (Eingabefeld Client)
  - Einer Testperson war nicht klar was unter Client einzutragen ist. Hier wird lediglich ein frei wählbarer Name für den zu startenden Testling vergeben um ihn innerhalb der Testsuite zu identifizieren.
10. Schnellstart Wizard sehr versteckt

- Der Wizard wurde von den Probanden erst sehr spät gefunden, nachdem die Einbindung des Testlings per Hand gescheitert ist.

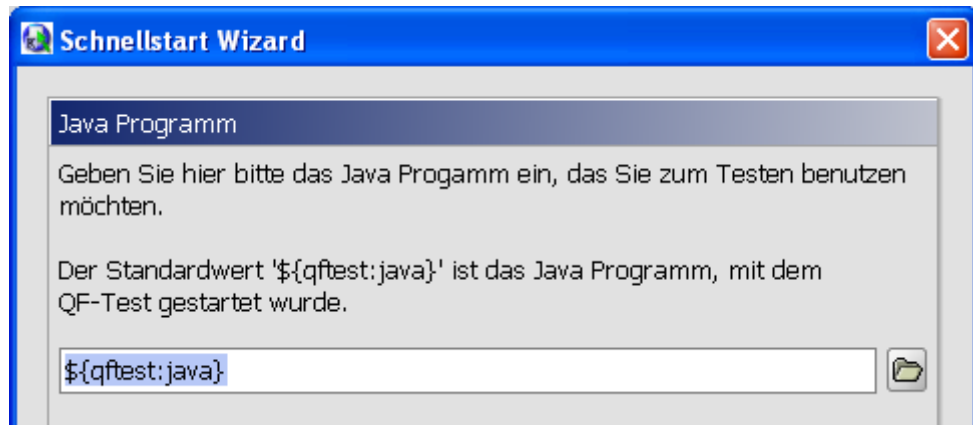


Abbildung 20: Verwirrender Wert in QF-Test

- Bei Java Anwendung „ausführbares Programm“ sehr verwirrend
  - Der voreingestellte Eintrag „`\$qftest:java`“ war für alle Probanden unverständlich. Alle haben hier versucht das JAR Archiv einzutragen. Es wird nicht deutlich, dass dies der Aufruf der JAVA VM ist.
- Label nicht im Komponentenfenster sichtbar
  - Das Label in dem das Ergebnis angezeigt wird, wird in der Komponentenliste nicht angezeigt. Auch die Eingabe des Namens der Komponente führt zu keinem Ergebnis.
- Auswahl einer Verknüpfung im Datei öffnen Fenster führt zu einem Fehler
  - Wird im Datei Öffnen Dialog eine Windows Verknüpfung (z.B. zu einem Ordner) geöffnet führt dies zu einem reproduzierbaren Fehler. Danach ist der Dialog nicht mehr verwendbar.
- Testperson ist verwundert, dass sie im Modus „Check Anlegen“ die GUI des Testlings nicht wie gewohnt verwenden kann.
- Stoppknopf sieht „ausgegraut“ aus
  - Testperson verwendete den Stoppknopf nicht, da er farblich aussieht wie eine nicht funktionierende Komponente.
- Testling ist nach dem Start nicht im Vordergrund
  - Testpersonen bemerkten zum Teil nicht, dass der Testling bereits läuft, weil der sich der Testling hinter der Anwendung befindet.
- Testwerkzeug erzeugt kein direktes Feedback beim Aufnehmen eines Tests
- Nach manueller Eingabe eines Checks einer Komponente, dessen ID nicht vorhanden ist, bleibt die ID im Check eingetragen obwohl sie offensichtlich falsch ist.

### 6.3.1.3 Verbesserungsvorschläge

Die größten Probleme hatten die Probanden offensichtlich bei der Einbindung des Testlings in die Testsuite. Eine prominenterer Platzierung des Schnellstart Wizards könnte hier durchaus Abhilfe schaffen, z.B. in Form eines Dialogs beim Öffnen einer neuen Testsuite.

Der Wizard selbst hat bei den Probanden insgesamt gute Dienste geleistet, allerdings sollte die Frage nach dem Java Programm (s. Abb. 7) überarbeitet werden. Hier sollte auf jeden Fall ergänzt werden, dass es sich hierbei um die Java VM handelt und nicht um den Testling. Zusätzlich könnte der Hinweis angebracht werden, dass dieser Eintrag in der Regel nicht verändert werden braucht.

Weiterhin könnte man auf die Frage nach dem Typ der Anwendung verzichten, indem man dem Benutzer einfach die Möglichkeit gibt, eine beliebige Datei auszuwählen. An deren Endung lässt sich nun recht einfach der Typ der Anwendung ableiten. Falls der Benutzer eine nicht unterstützte Datei auswählen würde, ließe sich immer noch eine passende Fehlermeldung ausgeben.

Ich empfehle hier ebenfalls eine Überarbeitung des Datei Öffnen Dialogs. Dieser wurde von allen Probanden als zu träge empfunden, so dass diese teilweise dachten, dass der Dialog nicht auf ihre Eingabe reagiert hat.

#### **6.3.1.4 Fazit**

Der Einstieg in dieses Werkzeug fiel allen Probanden sehr schwer, aber durch die im vorherigen Abschnitt empfohlenen Änderungen dürften diese leicht überwunden werden.

## 6.3.2 Froglogic Squish

### 6.3.2.1 Gelöste Aufgaben

Durch den automatisch erscheinenden Assistenten zur Einbindung des Testlings war dieser Teil der ersten Aufgabe bei allen Probanden in wenigen Minuten erledigt. Auch die Aufzeichnung von Benutzereingaben war bei allen erfolgreich. Als problematisch erwies sich hier das Anlegen von Checks (in diesem Tool VerificationPoints(VP) genannt).

Proband	Additionstest	Multiplikationstest	Aufgabe 2	Aufgabe 3	Zeit für Add-Test
1	✓	○	✗	✗	26 min
2	○	✗	✗	✗	-
3	✓	✓	○	✗	25 min

### 6.3.2.2 Erkannte Probleme

1. Feld „Test Data“ beim Anlegen eines neuen Tests
  - Keinem der Testpersonen war klar was man unter Test Data eintragen kann.
2. Anlegen von Checks sehr unhandlich

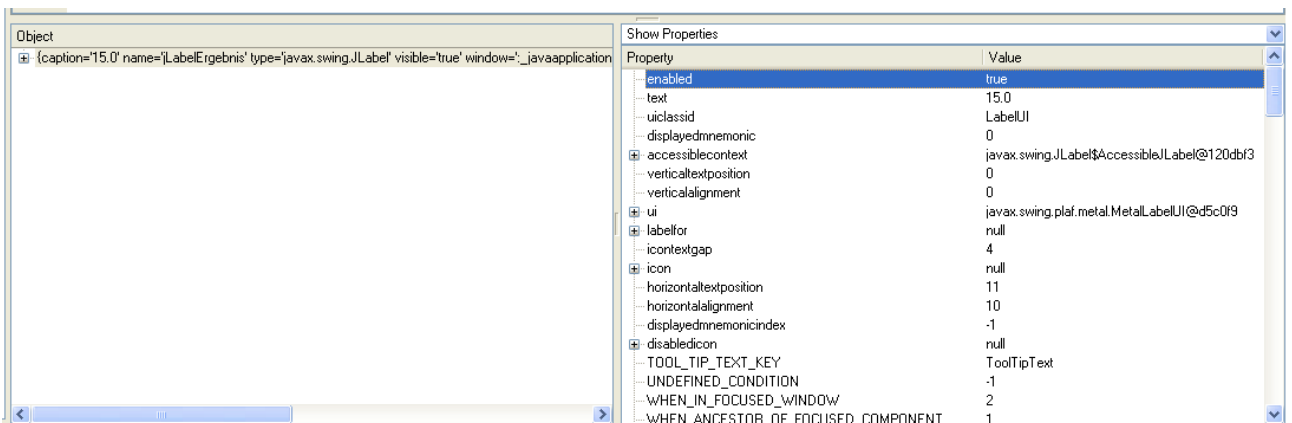


Abbildung 21: Modus 1 des Spy's

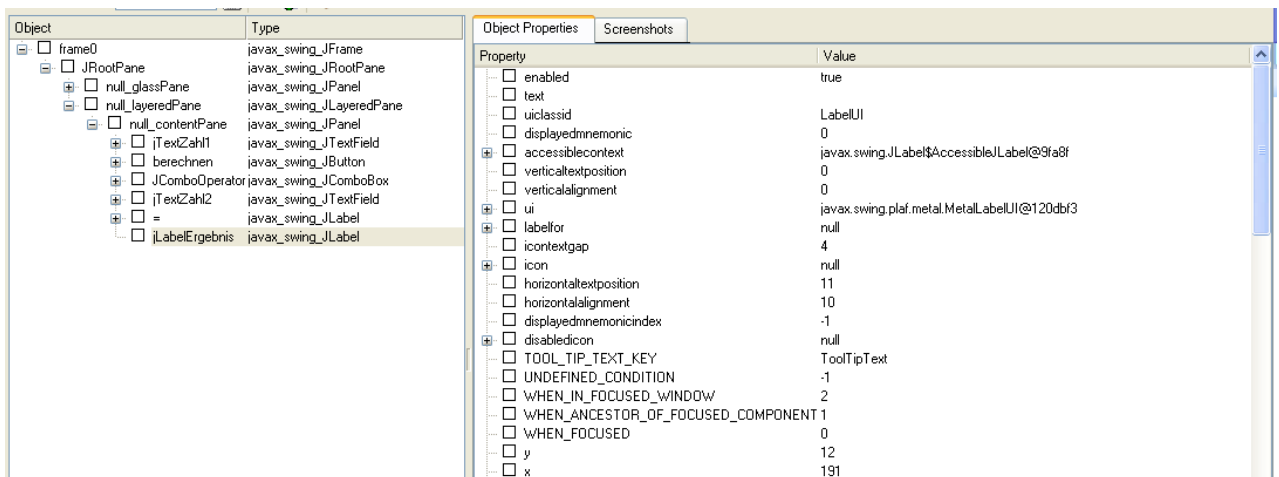


Abbildung 22: Modus 2 des Spy's

3. Zwei verschiedene Modi des Spy's verwirren die Probanden

- Im ersten Modus lassen sich die Eigenschaften der Komponenten nur ansehen aber nicht weiter verwenden. In Modus 2 (ausgelöst durch den Stopp an einem Breakpoint) hingegen, kann man mittels CheckBox eine Eigenschaft auswählen und als VP ins Testscript einfügen.
- 4. Keine direkte Einbindung der Komponenten aus dem Spy
  - Wird der Spy ohne Breakpoint verwendet, lassen sich die Eigenschaften die der Spy anzeigt nicht im Script verwenden.
- 5. Sinn von „add to object map“ im Spy
  - Diese Funktion gibt keine Rückmeldung und daher war den Probanden nicht klar, ob überhaupt etwas passiert ist.
- 6. Eine Testperson versuchte den Code für einen VP selbst zu schreiben, scheiterte aber
- 7. Es fehlt eine einfache Möglichkeit einen Test Case zu kopieren

### **6.3.2.3 Verbesserungsvorschläge**

Es ist sehr verwirrend, dass das Anlegen von VPs nur möglich ist, wenn das Testscript durch einen Breakpoint unterbrochen wird. Dies ist zwar in der Hilfe dokumentiert, wurde aber anfangs von allen Probanden ignoriert, da sie scheinbar keinen Zusammenhang zwischen VP und Breakpoint sahen.

Hier sollten zusätzliche Möglichkeiten geboten werden einen VP anzulegen. Beispielsweise wurde versucht eine VP dadurch zu erzeugen, dass die zu prüfende Eigenschaft aus dem Spy in das Testscript gezogen wurde (also mittels „Drag and Drop“).

### **6.3.2.4 Fazit**

Die Integration des Testlings direkt beim Anlegen der Testsuite zahlt sich aus. Hiermit hatte kein Proband Probleme, wie z.B. beim vorherigen Test. Auch die Aufzeichnung der Interaktionen mit der GUI bereitete den Probanden keine großen Probleme.

Als problematisch erwies sich bei allen Probanden das Anlegen der VP. Hier sollte dringend nachgebessert werden.

### 6.3.3 FEST

#### 6.3.3.1 Gelöste Aufgaben

Ein Problem bei diesem Test war die mangelnde Erfahrung aller Probanden mit Netbeans. Falls ich Probleme bemerkte half ich den Probanden umgehend. Desweiteren schien den Probanden nicht klar zu sein, dass das erwähnte Plug-in keinerlei Schaltflächen etc. in Netbeans integriert.

Nichtsdestotrotz waren zwei Probanden mit diesem Werkzeug am Erfolgreichsten.

Proband	Additionstest	Multiplikationstest	Aufgabe 2	Aufgabe 3	Zeit für Add-Test
1	✓	○	✗	✗	19 min
2	✓	✓	○	✗	19 min
3	✓	✓	✓	✓	18 min

#### 6.3.3.2 Erkannte Probleme

1. Bedeutung von FrameFixture
  - Die Probanden erkannten anfangs nicht, dass FrameFixture den Zugriff auf den Frame des Testlings bietet. Teilweise wurde versucht den Frame alleine zu starten, so war es FEST natürlich nicht möglich auf die Komponentn der GUI zuzugreifen.
2. Probanden fehlte die genaue Kenntnis des Testlings
  - Die Probanden mussten mehrfach im Code des Testlings nach dem Namen der GUI Komponenten suchen.
3. Debugging schwierig
  - Erst nachdem ich die Probanden auf die Debugging-Ausgaben hingewiesen habe, fanden sie die Fehler.
4. Direkte Auswahl des Operators über String teilweise nicht möglich
  - Wird versucht den „\*“ oder „/“ Operator als String auszuwählen, schlägt der Test reproduzierbar fehl. Hier muss eine Auswahl mittels Index gewählt werden.
5. Einsetzen von Wissen aus JUnit
  - Ein Proband setzte sein Wissen aus JUnit ein und versuchte die Vergleiche mittels Assert durchzuführen, da dies nicht vorgesehen war der Testfall nicht erfolgreich.

#### 6.3.3.3 Verbesserungsvorschläge

Das Hauptproblem bei den Probanden bestand darin, die Fehlermeldungen zu verstehen. Diese sollten wesentlich übersichtlicher gestaltet werden.

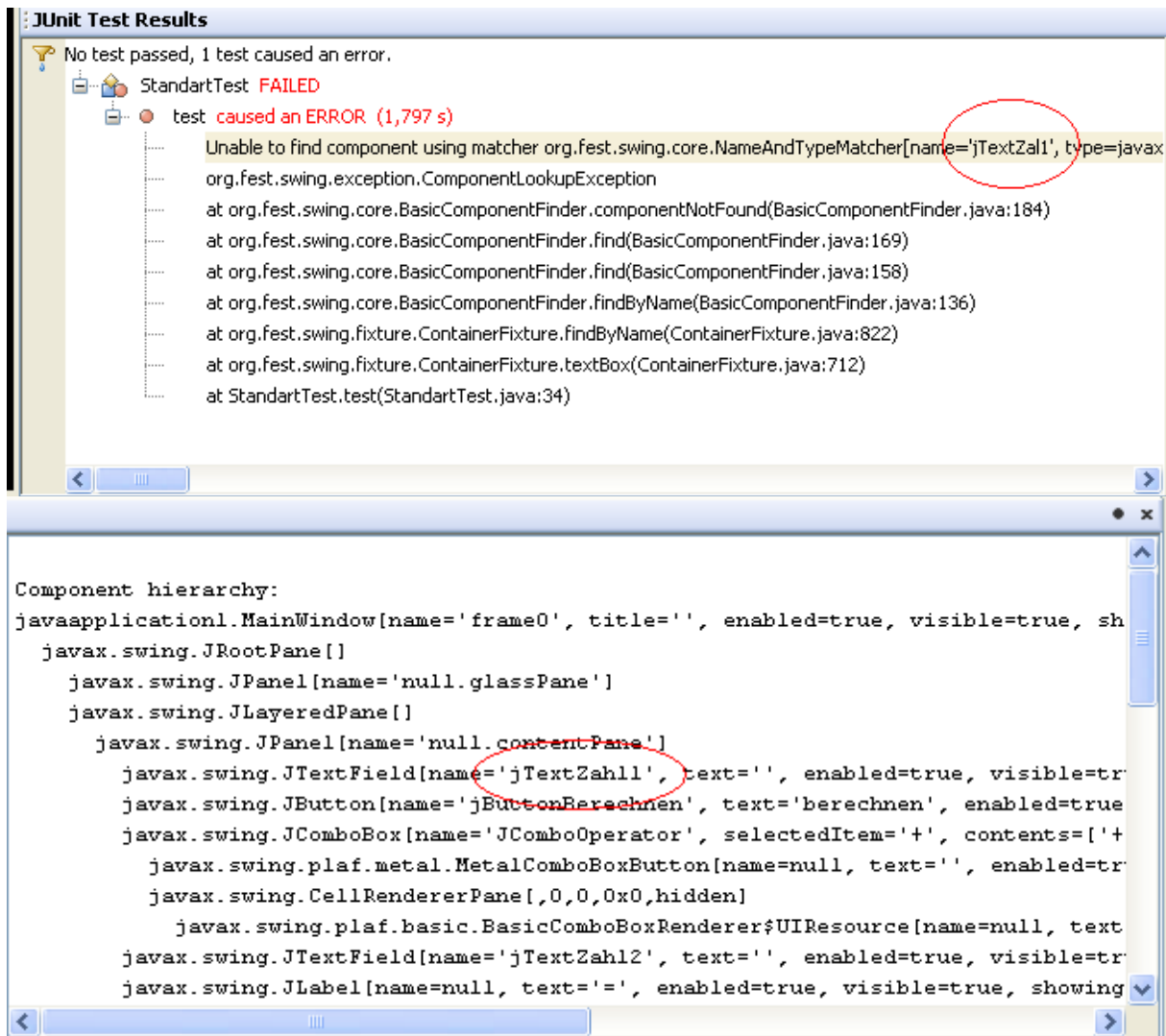


Abbildung 23: Debugging in FEST

Oben sieht man die Ausgabe von FEST bei einem Fehler. Ich habe im Quelltext des Tests versucht auf die Komponente „jTextZahl1“ zuzugreifen, wie es bei einem Tippfehler schnell passieren kann. Im oberen Fenster steht, dass der Test fehlgeschlagen ist und in der markierten Zeile, dass der Matcher die Komponente mit dem Namen „jTextZahl1“ nicht finden konnte. Ein Blick in das untere Fenster zeigt die Komponentenhierarchie des Frames. Bei der überschaubaren GUI des Testlings kann man hier nach kurzem Suchen herausfinden, dass es eine Komponente mit dem Namen „jTextZahl2“ gibt.

Somit unterstützt FEST den Anwender bei Debugging, aber es ist insgesamt eine sehr mühsame Arbeit. Übersichtlicher wären hier beispielsweise angepasste Ausgaben je nach Art des Fehlers. In diesem Fall würde es auch reichen die Komponentenhierarchie kompakter darzustellen, so dass nur der Name und der Typ der Komponente angezeigt werden würde.

...	
javax.swing.JTextField	jTextZahl1
javax.swing.JButton	jButtonBerechnen
javax.swing.JComboBox	JComboOperator
...	

In dieser Aufstellung ließe sich viel schneller die gesuchte Komponente finden.

#### **6.3.3.4 Fazit**

Die Nähe zu den gewohnten JUnit Tests machte es allen Probanden relativ einfach den Einstieg zu finden. Nachdem Sie das Konzept verstanden haben, konnten sie sehr schnell eigene Tests schreiben. Ausgebremst wurde Sie hauptsächlich dadurch, dass Sie die Namen der Komponenten des Testlings nachschlagen müssen und das Debugging des Testscripts leider recht zeitaufwendig ist. Außerdem wäre eine übersichtliche Auflistung der unterstützten Komponenten praktisch gewesen.



## 6.4 Mögliche Erklärung zum schlechten Abschneiden von QF-Test

Das schlechte Abschneiden des ersten Werkzeugs lässt sich hauptsächlich dadurch erklären, dass die Probanden bis dahin noch nie ein solches Werkzeug verwendet haben (s. Kap. 6.2: Auswertung der Fragebögen). Somit war den Probanden nicht klar, dass sie mit den Werkzeugen den Testling öffnen können und die Komponenten direkt per Klick in das Testscript einfügen können. Beim zweiten Test war dieser Umstand den Probanden bekannt und dies änderte auch deren Herangehensweise. Sie suchten gleich einen Weg den Testling einzubinden und mit dem Werkzeug zu öffnen, um die Komponenten anklicken zu können.

Dies ist allerdings nur ein kleiner Aspekt, dessen Gewicht sich nicht genau beziffern lässt. Hauptgrund ist meiner Meinung nach der versteckte Schnellstart Wizard, sowie der verwirrende Eintrag unter dem Java Programm.

## 6.5 Erkenntnisse aus den Tests

Sehr auffällig ist, dass das open-Source Werkzeug insgesamt am Besten abgeschnitten hat. Dies bestätigt die Annahme, dass sich die Werkzeuge an verschiedene Personengruppen richten. Alle Testpersonen sind am ehesten als Programmierer anzusehen. Somit kamen sie auch mit dem Werkzeug am Besten zurecht, dass sich augenscheinlich auch an Programmierer richtet. Das Anlegen von Tests in FEST ähnelt der Arbeit des Programmierens sehr. Dies wird auch dadurch unterstützt, dass die Tests in der gewohnten IDE angelegt werden können (z.B. Netbeans oder Eclipse).

Die kommerziellen Werkzeuge besitzen eine eigene GUI und zeigen so schon einen gewissen Abstand, sprich der Programmierer muss seine gewohnte IDE verlassen. Zusätzlich muss sich der Programmierer auf ein völlig neues Tool einlassen, dies bereitete allen Probanden scheinbar große Probleme.

Allerdings denke ich, dass nach dem Überwinden der anfänglichen Hürden, der Umgang mit den Werkzeugen kein Problem mehr darstellen sollte. Die kommerziellen Tools erlauben den Einstieg ohne weitere Programmierkenntnisse, dies ist für Programmierer ungewohnt.

## 7 Vergleich aller Werkzeuge

### 7.1 Kommerziell

Schon auf den ersten Blick fällt auf, dass eigenständige Programme den kommerziellen Bereich beherrschen. Komponenten lassen sich einfach per Klick auf der GUI in die Testscripte einfügen. Der Einstieg in solche Werkzeuge fällt somit vor allem nicht-Programmierern leicht, da so gut wie keine Kenntnisse über den Aufbau von GUIs nötig sind. Assistenten erleichtern den Benutzern die Einbindung des Testlings.

Es lässt sich allerdings erkennen, dass die Probanden trotzdem nicht alle Aufgaben mit diesen Werkzeugen lösen konnten. Offensichtlich sind diese Werkzeuge für Tester entwickelt worden und nicht direkt für Programmierer.

## 7.2 Frei

Im freien Bereich sind die meisten Werkzeuge an JUnit angelehnt. Sie besitzen i. d. R. keine eigene GUI, somit müssen die Testscripte per Hand verfasst werden. Dazu ist die Kenntnis der GUI-Komponentenamen absolut notwendig. Allerdings ließ sich beobachten, dass es den Probanden einen einfacheren Einstieg gewährt. Dies ist wohl darauf zurückzuführen, dass sich die Tests sehr stark an JUnit anlehnen und auch der Arbeit eines Programmierers sehr ähnlich sind.

### 7.3 Fazit

Einer der größten augenscheinlichen Unterschiede zwischen den kommerziellen und den freien Werkzeugen besteht in der Benutzeroberfläche. Im kommerziellen Bereich wird bei allen drei Produkten eine eigene grafische Oberfläche angeboten, über die die Tests angelegt werden. Innerhalb dieser Umgebungen konnte dann der Testling gestartet werden und es ließen sich direkt die Komponenten im Testling anklicken die z.B. abgefragt werden sollen.

Bei den freien Programmen werden FEST und jfcUnit komplett ohne eine eigene GUI ausgeliefert. Die Erstellung der Tests ist hiermit etwas mühsamer, da sie nicht einfach „zusammengeklickt“ werden können, sondern der Benutzer muss die Testscripte komplett selbst schreiben und immer die Namen aller Komponenten wissen.

Es ließ sich erkennen, dass der Einstieg für einen Programmierer leichter ist, wenn sich das Testtool am Programmierer-Alltag orientiert, wie in diesem Fall FEST. Der Einstieg in die Werkzeuge mit eigener GUI fiel demgegenüber schwerer.

## 8 Ausblick

Die anfängliche Hürde zum Einstieg in die automatisierten GUI Tests ist nicht sehr hoch. Die Testkandidaten waren alle in der Lage innerhalb kürzester Zeit einfache Tests zu Erstellen, die viele grobe Fehler in den Testlingen finden können.

Weiterhin haben die Tests gezeigt, dass auch mit den frei verfügbaren Werkzeugen schnell einfache Tests erstellt werden können, ohne eine hohe Investition in Software oder Schulung investieren zu müssen.

## 9 Quellen

FEST. (kein Datum). *FEST Documentation*. Abgerufen am 12. Oktober 2008 von <http://fest.easytesting.org/swing/wiki/pmwiki.php?n=FEST-Swing.GettingStarted>

Froglogic. (kein Datum). *Froglogic - Squish*. Abgerufen am 20. August 2008 von <http://www.froglogic.com/pg?id=Products&category=squish&sub=overview&subsub=overview>

jfcUnit. (kein Datum). *jfcUnit*. Abgerufen am 2. Oktober 2008 von <http://jfcunit.sourceforge.net/>

Liggemeyer, P. (2000). *Qualitätssicherung softwareintensiver technischer Systeme*. Spektrum Akademischer Verlag.

Pounder. (kein Datum). *Pounder*. Abgerufen am 2008. September 22 von <http://pounder.sourceforge.net/download.php>

*Praxis der Wirtschaftsinformatik 2000 - Usability Engeneering*.

QFS. (kein Datum). *Quaility First Software GmbH*. Abgerufen am 5. Oktober 2008 von <http://www.qfs.de/>

Rampl, D. I. (kein Datum). *Handbuch Usability*. Abgerufen am 3. Dezember 2008 von <http://www.handbuch-usability.de>

Spillner, A., & Linz, T. (2005). *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH.

Testing, F. -F. (kein Datum). *FEST Documentation*. Abgerufen am 20. August 2008 von <http://fest.easytesting.org/swing/wiki/pmwiki.php?n=Main.HomePage>

## 10 Anhang

### 10.1 Abbildungsverzeichnis

Abbildung 1: Froglogic Squish .....	10
Abbildung 2: QF-Test .....	12
Abbildung 3: Ein FEST-Test in Netbeans.....	14
Abbildung 4: Pounder.....	15
Abbildung 5: GUI des Testlings.....	17
Abbildung 6: Hinweisdialog des Testlings bei negativen Ergebnis.....	17
Abbildung 7: Begrüßungsbildschirm von Squish .....	19
Abbildung 8: Squish Control Bar.....	20
Abbildung 9: Breakpoint im Testscript zum Anlegen des VP .....	20
Abbildung 10: Attribute des Labels im Spy .....	21
Abbildung 11: Kompletter Additionstest inklusive VP .....	21
Abbildung 12: Lösung der zweiten Aufgabe mit Squish.....	22
Abbildung 13: Test Log nach der dritten Aufgabe.....	22
Abbildung 14: Neue Testsuite mit integriertem Testling .....	23
Abbildung 15: Testsequenz des Additionstests.....	24
Abbildung 16: Additionstest in QF-Test .....	24
Abbildung 17: Video aus dem Usability Labor.....	29
Abbildung 18: Auswahl des Typs der Anwendung in QF-Test.....	33
Abbildung 19: Undeutliche Fehlermeldung in QF-Test .....	33
Abbildung 20: Verwirrender Wert in QF-Test .....	34
Abbildung 21: Modus 1 des Spy's.....	36
Abbildung 22: Modus 2 des Spy's.....	36
Abbildung 23: Debugging in FEST .....	39

## 10.2 Fragebogen

### Fragebogen für die Probanden der Usability Tests

Name:

**1) Haben sie jemals ein Testautomatisierungswerkzeug genutzt? (z.B. JUnit etc)**

Ja

Nein

**2) Haben sie jemals ein Testautomatisierungswerkzeug für grafische Benutzeroberflächen genutzt? Wenn ja, welche?**

keine

Ja, nämlich: \_\_\_\_\_

**3) Wie verhalten Sie sich direkt nach dem Start einer Ihnen unbekanntem Software?**

Ich suche die Antwort auf meine Fragen bei der Bedienung im Internet

Ich schaue in die Hilfe

Ich versuche mich erstmal selbst zurecht zu finden



## 10.3 Aufgabenzettel

### Aufgabenzettel für die Probanden des Usability Tests

Im Folgenden erhalten Sie die Aufgaben die Sie mit Hilfe der Werkzeuge lösen sollen. Bitte lösen sie mit jedem Werkzeug alle Aufgaben!

Die gegebenen Werkzeuge ermöglichen es ihnen automatisierte GUI-Tests auszuführen. Dazu sind sie in der Lage scriptgesteuert mit der GUI des Testlings (das Programm dass getestet werden soll) zu interagieren. Des Weiteren ist es möglich die GUI abzufragen z.B. den Inhalt einer Textbox etc.

Der Testling ist ein sehr einfaches Java Programm mit einer rudimentären grafischen Oberfläche, die auf Swing basiert. Das Programm liegt im Quellcode wie auch als kompiliertes JAR vor und ist bereits in Netbeans geöffnet. Es realisiert einen sehr einfachen Taschenrechner. Zusätzlich wird ein Hinweisdialog geöffnet, falls das Ergebnis einer Berechnung negativ ist. Bitte machen sie sich an dieser Stelle kurz mit dem Testling vertraut.

Versuchen sie zuerst die Aufgaben spontan zu lösen, falls dies nicht funktioniert dürfen sie natürlich gerne die Hilfefunktion sowie die zur Verfügung gestellten Tutorials verwenden.

Bitte beschreiben sie laut was sie gerade versuchen und gegebenenfalls auf welche Probleme sie dabei stoßen. Dies ist sehr hilfreich für die spätere Auswertung der Tests.

### Aufgaben

- 1) Erstellen sie jeweils einen Testfall für Additions- und den Multiplikationsoperator! Realsieiren sie also eine automatische Eingabe der Operanden, eine Auswahl des passenden Operators sowie den Klick auf den „Berechnen“ Button. Zusätzlich soll das Ergebnis der Berechnung überprüft werden. Sie können dabei beliebige Zahlen verwenden, achten sie aber darauf nur positive Ergebnisse zu generieren. Führen sie beide Testfälle aus und stellen sie sicher dass alle fehlerfrei durchgeführt werden konnten.
- 2) Erstellen sie einen Testfall bei dem ein negativen Ergebnis entsteht! Verwenden Sie dafür den Minusoperator. Ihr Testfall soll zum Einen prüfen ob die Warnmeldung wie gewünscht erscheint (diese anschließend mit „OK“ bestätigen) und zum Anderen auch das Ergebnis der Berechnung prüfen. Prüfen sie auch hier ob der Testfall fehlerfrei durchgeführt werden konnte.
- 3) Ändern sie im Quellcode des Testlings in der Methode „berechne“ die Berechnung der Summe, sodass ein falsches Ergebnis berechnet wird! Kompilieren sie die Anwendung ggf. neu und starten Sie anschließend einen Testfall aus Aufgabe 1 erneut. Stellen sie sicher, dass dieser nun fehlschlägt.

## 10.4 Schnellstartanleitung für FEST<sup>6</sup>

### Getting Started

FEST (Fixtures for Easy Software Testing) is an open source project ([Apache 2.0 license](#)) that aims at making software testing simpler. This guide explains how to get started with FEST's Swing module, which makes creation and maintenance of robust GUI functional tests easy.

### Before you start

Before starting to write any test, please do the following:

1. Download the latest version of the Swing module from the [project's download page](#). The filename should be similar to `fest-swing-{VERSION}.zip`, where `{VERSION}` is the latest version of the module.
2. Include the file `fest-swing-{VERSION}.jar` and its dependencies (jar/zip files, located in the 'lib' folder in the downloaded zip file) in your classpath.

### Writing your first GUI test

When writing GUI tests, please use the fixtures in the package `org.fest.swing.fixture`. These fixtures provide specific methods to simulate user interaction with a GUI component and also provide assertion methods that verify the state of such GUI component. Although you could work with the FEST Robot (`org.fest.swing.core.Robot`) directly, the `Robot` is too low-level and requires considerably more code than the fixtures.

There is one fixture per Swing component. Each fixture has the same name as the Swing component they can handle ending with "Fixture." For example, a `JButtonFixture` knows how to simulate user interaction and verify state of a `JButton`.

For our first test, let's assume we have a very simple `JFrame` that contains a `JTextField`, a `JLabel` and a `JButton`. The expected behavior of this GUI is: when user clicks on the `JButton`, the text of the `JTextField` should be copied to the `JLabel`.

The following sections describe the steps necessary to test our GUI.

#### 1. Create a fixture for a Frame or Dialog

---

<sup>6</sup> <http://fest.easytesting.org/swing/wiki/pmwiki.php?n=FEST-Swing.GettingStarted> abgerufen am 12. November 2008

Create a fixture to handle either a `Frame` or a `Dialog` (depending on the GUI to test) in the "setUp" method of your test. The "setUp" method is the method that initializes the test fixture before running each test method:

- `setUp` method (JUnit 3.8.x)
- any method marked with `@Before` (JUnit 4.x)
- any method marked with `@BeforeMethod` (TestNG)

Example:

```
private FrameFixture window;

@BeforeMethod public void setUp() {
    window = new FrameFixture(new MyFrame());
    window.show(); // shows the frame to test
}
```

## 2. Clean up resources used by FEST-Swing

FEST-Swing forces sequential test execution, regardless of the testing framework (JUnit or TestNG.) To do so, it uses a semaphore to give access to the keyboard and mouse to a single test. Clean up resources after running each test method to releases the lock on such semaphore. To clean up resources simply call the method `cleanUp` in the FEST-Swing fixture inside:

- `tearDown` method (JUnit 3.8.x)
- any method marked with `@After` (JUnit 4.x)
- any method marked with `@AfterMethod` (TestNG)

Example:

```
@AfterMethod public void tearDown() {
    window.cleanUp();
}
```

## 3. Write methods to test the GUI behavior

Start using the FEST-Swing fixture to test your GUI. FEST-Swing fixtures simulate a user interacting with a GUI in order to verify that such GUI behaves as we expect. For our example, we need to verify that the text in the `JTextField` is copied to the `JLabel` when the `JButton` is clicked:

```
@Test public void shouldCopyTextInLabelWhenClickingButton() {
    window.textBox("textToCopy").enterText("Some random text");
    window.button("copyButton").click();
    window.label("copiedText").requireText("Some random text");
}
```

## Putting everything together

The following code listing shows the whole test that verifies the described GUI is behaving correctly:

```
import org.testng.annotations.*;
import org.fest.swing.fixture.FrameFixture;

public class FirstGUITest {

    private FrameFixture window;

    @BeforeMethod public void setUp() {
        window = new FrameFixture(new MyFrame());
        window.show(); // shows the frame to test
    }

    @AfterMethod public void tearDown() {
        window.cleanUp();
    }

    @Test public void shouldCopyTextInLabelWhenClickingButton() {
        window.textBox("textToCopy").enterText("Some random text");
        window.button("copyButton").click();
        window.label("copiedText").requireText("Some random text");
    }
}
```

## 10.5 Testmetriken

### 10.5.1 Squish

Proband	Linksklicks	Rechtsklicks	Mausstrecke in m	Tastaturanschläge
1	248	18	63	34
2	540	25	57	145
3	359	23	92	124

### 10.5.2 QF-Test

Proband	Linksklicks	Rechtsklicks	Mausstrecke in m	Tastaturanschläge
1	286	7	48	172
2	555	13	55	82
3	428	33	82	34

### 10.5.3 FEST

Proband	Linksklicks	Rechtsklicks	Mausstrecke in m	Tastaturanschläge
1	196	24	42	335
2	371	22	36	605
3	296	25	50	446

## 10.6 Glossar

Testling	Der Testling ist das zu testende Programm
SUT	System under Test; s. Testling
GUI	Graphical User Interface – grafische Benutzeroberfläche eines Programms
nightly build	Nächtliche Kompilierung eines Projekts; i.d.R. automatisiert
Testsuite	Eine Sammlung von Tests

## Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw. §24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

X

---

Benjamin Sass

Kisdorf, 16. Januar 2009